TURING 图灵程序设计丛书 数据库系列

Addison Wesley

MySQL Fourth Edition

MySQL技术内幕

（第4版）

[美] Paul DuBois 著
杨晓云 王建桥 杨涛 译

- MySQL权威专家力作
- 全面、深入、细致入微
- 一册在手，别无他求



人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Paul DuBois

Sun公司MySQL文档团队的技术作者、开源社区和MySQL社区活跃的技术专家，同时也是一名数据库管理员。他曾参与过MySQL在线文档的编写工作。除本书外，他还著有*MySQL and Perl for the Web*、*MySQL Cookbook*、*Using csh and tcsh*以及*Software Portability with imake*等书。

TURING 图灵程序设计丛书 数据库系列

MySQL Fourth Edition

MySQL技术内幕
（第4版）

[美] Paul DuBois 著
杨晓云 王建桥 杨涛 译



人 民 邮 电 出 版 社
北 京

图书在版编目（C I P）数据

MySQL技术内幕 ：第4版 / （美）杜波依斯
(DuBois,P.）著 ；杨晓云，王建桥，杨涛译．-- 北京 ：
人民邮电出版社，2011.7
（图灵程序设计丛书）
书名原文：MySQL，Fourth Edition
ISBN 978-7-115-25595-2

Ⅰ．①M… Ⅱ．①杜… ②杨… ③王… ④杨… Ⅲ．①
关系数据库－数据库管理系统，MySQL Ⅳ．
①TP311.138

中国版本图书馆CIP数据核字(2011)第101759号

内 容 提 要

本书介绍了MySQL的基础知识及其有别于其他数据库系统的独特功能，包括SQL的工作原理和MySQL API的相关知识；讲述了如何将MySQL与Perl或PHP等语言结合起来，为数据库查询结果生成动态Web页面，如何编写MySQL数据访问程序；详细讨论了数据库管理和维护、数据目录的组织和内容、访问控制、安全连接等。附录还提供了软件的安装信息，罗列了MySQL数据类型、函数、变量、语法、程序、API等重要细节。

本书是一部全面的MySQL指南，对数据库系统感兴趣的读者都能从中获益。

图灵程序设计丛书

MySQL技术内幕（第4版）

◆ 著　[美] Paul DuBois
译　杨晓云　王建桥　杨　涛
责任编辑　王军花
执行编辑　谢灵芝

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：56.75
字数：1 519千字　2011年7月第1版
印数：1－3 000册　2011年7月北京第1次印刷

著作权合同登记号　图字：01-2008-5832号

ISBN 978-7-115-25595-2

定价：139.00元

读者服务热线：(010)51095186转604　印装质量热线：(010)67129223

反盗版热线：(010)67171154

版 权 声 明

致　　谢

针对本书的各个版本，下面对相关人士表示感谢。

第 4 版

本书第4版的技术审稿人Stephen Frein和Tim Boronczyk发现了许多需要纠正或澄清的地方，Ulf Wendel和Johannes Schlüter对与PHP有关的内容给出了修改意见，我要感谢他们当中的每一个人。

参与本书第4版出版工作的Pearson出版公司的员工有策划编辑Mark Taber、执行编辑Michael Thurston、项目编辑Jovana San Nicolas-Shirley、排版员Jake McFarland、索引负责人Cheryl Lenser和封面设计者Gary Adair。

还有我的妻子Karen，我要感谢她对我长期伏案写作的鼓励和支持。

第 3 版

本书第3版由Zak Greant和Chris Newman做了仔细的技术审查，他们的努力在许多方面改善了本书的原稿。MySQL专家Monty和MySQL AB公司的研发人员在我向他们咨询时提供了许多真知灼见。

参与本书第3版出版工作的Pearson出版公司的员工有策划编辑Shelley Johnston、执行编辑Damon Jordan和项目编辑Andy Beaster。

我还要再次感谢我妻子Karen长期以来对我反复修改和完善本书的理解和支持。

第 2 版

对于本书第2版，技术审稿人在发现、纠正和澄清错讹方面再次起到了至关重要的作用。Hang Lau和Shane Kirk是该版本的技术审稿人。我还要感谢Monty Widenius、Alexander Barkov、Jani Tolonen等MySQL研发人员耐心解答我的许多技术问题，他们提供的答案使本书增色不少。

参与本书第2版出版工作的New Riders出版社的员工有副社长Stephanie Wall、执行编辑Chris Zahn、高级项目编辑Lori Lyons、文字编辑Pat Kinyon、索引负责人Cheryl Lenser和排版员Stacey Richwine-DeRome。

还有，像往常一样，如果没有我妻子Karen在背后支持我（虽然本书的读者看不到），本书将大为逊色。

第 1 版

感谢以下技术审稿人对本书的评论、批评和指正：David Axmark、Vijay Chaugule、Chad Cunningham、Bill Gerrard、Jijo George John、Fred Read、Egon Schmid和Jani Tolonen。我还要特别感谢MySQL的主要创始人——人称Monty的Michael Widenius，他不仅细心地审阅了本书的稿件，还耐心

解答了我在本书写作过程中向他提出的数以百计的问题。如果读者还在书里发现了错漏，就只能怪我本人学艺不精了。我还要感谢Tomas Karlsson、Colin McKinnon、Sasha Pachev、Eric Savage、Derick H. Siddoway和Bob Worthy，他们对这本书进行了认真的校对，并帮我把本书改进成现在的样子。

我还要衷心感谢New Riders出版社的有关工作人员，他们构思本书，并使它成型。Laurie Petrycki是编辑主任。Katie Purdum是本书的策划编辑，她一直关心着本书的写作，并不断督促我抓紧时间完成。Leah Williams既是开发编辑又是文字编辑，她为本书加了不少的班，尤其是在本书出版的最后阶段。本书的索引是由Cheryl Lenser和Tim Wright制作的。本书的项目编辑John Rahm也为本书倾注了大量的心血。Debra Neel负责校对，Gina Rexrode和Wil Cruz负责排版。在此谨向以上人员表示我衷心的感谢。

最后，我还要感谢我的妻子Karen。为了支持我的工作，她推迟了她自己的一本书的写作和出版计划。正是有了她的耐心和理解，我才能整天埋头于写作当中。没有她的支持，本书就不能如此顺利完成，可以说，这本书里的每一页都有她的贡献。

联系我们

作为本书的读者，你的意见和看法将是最为重要的。希望大家能够不吝赐教，告诉我们哪些地方做得不错，哪些地方还需要改进，哪些东西是你们想知道而没有收录在本书里的。总之，只要是读者的声音，我们都将认真倾听。

大家可以通过电子邮件或普通信件直接与我联系，好让我了解你们对本书的看法以及改进建议。

不过，对于大家在学习本书时遇到的技术性问题，我不一定能帮得上忙，因为我每天都会收到大量邮件，可能无法一一回复每封邮件。

请在邮件中把本书的书名、作者、你的姓名、电话或者电子邮件地址写清楚。我将认真对待各位读者的来信并与本书的作者和编辑人员交流。下面是我们的联系方式：

电子邮件：feedback@developers-library.info

普通信件：Mark Taber
Associate Publisher
Pearson Education
800 East 96th Street
Indianapolis, IN 46240 USA

读者服务

请到网站informit.com/register本书主页注册以获得本书的更新信息、可下载的资源和勘误。

前　　言

无论是在商业、科研和教育等方面的传统性应用项目里，还是作为因特网搜索引擎的后端支持，RDBMS（Relational Database Management System，关系数据库管理系统）在许多场合都是一种极其重要的工具。良好的数据库系统对于管理和访问信息资源来说至关重要，但很多企事业单位都没有足够的财力建立起自己的数据库系统。从历史上看，数据库系统一直是价格昂贵的产品，无论是软件本身还是后续的技术支持，供货商从来都是漫天要价。此外，为了获得令人满意的性能表现，数据库引擎往往对计算机硬件要求很高，而这又将使数据库系统的运营成本大大增加。

计算机硬件和软件在最近几年里的发展已经使这种情况得到了改善。小型桌面系统和服务器的价格越来越低，性能越来越高，而为它们编写高性能操作系统正成为一种潮流。这些操作系统有的可以从因特网免费获得，有的可以通过价格低廉的CD获得。它们包括BSD Unix操作系统的几种变体（如FreeBSD、NetBSD、OpenBSD等）以及各种各样的Linux发行版本（如Fedora、Debian、Gentoo、SuSE等）。

免费的操作系统因诸如GNU C编译器`gcc`之类的免费开发工具的发展而日臻完善。让任何人都能得到想要的软件，这正是*开源*运动的一部分。开源项目已经为我们提供了很多重要的软件产品，如因特网上使用范围最广的Web服务器Apache，以及广泛应用的通用脚本语言Perl、Python和Ruby，还有非常便于编写动态Web页面的PHP语言等。与此形成鲜明对比的是，如果决定采用某种专有的商业化解决方案，就不得不忍受供货商漫天要价，而且还极有可能根本看不到它的源代码。

开源运动也使免费的数据库软件和数据库系统越来越容易获得。例如MySQL就是一种免费的数据库系统，它是一种客户/服务器模式的关系数据库管理系统，最初起源于欧洲的斯堪的纳维亚半岛。MySQL由以下组件构成：一个SQL服务器、一些用来访问该服务器的客户程序、一套用来对数据库进行管理的软件工具，以及供用户自己编写程序的编程接口。

MySQL起源于Michael Widenius（外号Monty）在1979年为瑞典的TcX公司开发的一套名为UNIREG的数据库工具。到了1994年，TcX公司开始寻求一种能够用来开发Web应用的数据库服务器。TcX公司对几种商业化的服务器进行了测试，对它们在处理TcX公司的大数据表时的速度都不太满意。该公司还测试了mSQL，它缺少某些必要的功能。因此，Monty开始开发一种新的服务器。因为mSQL有一些免费的软件工具，所以新服务器的编程接口被有意设计成与mSQL所使用的编程接口非常相似。采用相似的编程接口将大大减少把那些免费的软件工具移植到MySQL的工作量。

到了1995年，Detron HB公司的David Axmark开始在因特网上推广和发行TcX公司研发的MySQL。David为MySQL编写了许多文档，增加了利用GNU组织的`configure`工具进行安装配置的功能。适用于Linux和Solaris系统的MySQL 3.11.1的二进制版本于1996年面世。如今，MySQL不仅能够在许多种计算机平台上运行，还同时提供二进制版本和源代码版本。MySQL在开源许可证和商业许可证下的发布、技术支持、监控服务和培训工作以前由MySQL AB公司专门负责。Sun公司在2008年收购了MySQL

AB公司，但保持了MySQL的开源特色（Sun公司的许多产品现在都可以在开源许可证下获得和使用）。

早期的MySQL广受欢迎的主要原因是它的速度和简单性，但因为缺少诸如事务处理（transaction）和外键支持（foreign key support）之类的高级功能，所以也有一些批评的声音。MySQL的开发和完善工作从未停止，发展至今，事务处理、外键支持、复制（replication）、子查询、存储过程、视图和触发器等功能都已被添至其中。这些功能让MySQL进入了企业级数据库软件的行列。结果，许多原来只考虑大型数据库系统而对MySQL不屑一顾的用户开始认真评估MySQL了。

MySQL的可移植性非常好，它可以运行在商业化操作系统（如Mac OS X、HP-UX和Windows）以及包括桌面电脑和企业级服务器的硬件平台上。此外，MySQL的运行性能绝不逊色于任何一种数据库系统，即使面对容纳着几十亿条数据记录的大型数据库，它也能游刃有余。在商业领域里，MySQL的地盘一直在扩大，这是因为许多公司的老板发现，与购买商业化许可证和技术支持服务相比，只需花一点零头就可以满足数据库处理需求。

未来我们将可以在功能强大但价格低廉的硬件设备上运行免费的操作系统，将有越来越多的人和商业机构在各种各样的硬件系统上拥有强大的计算能力和其他功能，MySQL则在其中起着重要作用。获得强大计算能力的经济成本的门槛正变得越来越低，大型数据库解决方案对普通用户和企业来说也已经不再是可望不可及的了。在过去，高性能的RDBMS只能出现在广大中小企业的梦想里，可现在，只需付出极低的成本和代价就能享用到这些东西。这一点对个人用户而言就更加突出了。就拿我本人来说吧，我有一台苹果笔记本电脑，在它的Mac OS X操作系统上，我同时使用着MySQL以及Perl、Apache和PHP。这使我能够随时随地工作，而这一解决方案的总成本只是笔记本电脑的价钱而已。

为什么要选用MySQL

如果你正在寻求一种完全免费或者价格比较低廉的数据库管理系统，可以从MySQL、PostgreSQL和SQLite等软件中选择一个。在对MySQL和其他数据库系统进行评估之前，首先要弄清楚什么因素对自己最重要。你需要从运行性能、技术支持、特色功能（例如与SQL的兼容程度和可扩展性等）、许可证条件、购买价格等多方面进行全面的考虑。由此判断，MySQL在以下方面有比较吸引人的优势。

- **运行速度**。MySQL的运行速度相当快，MySQL开发人员相信它是目前最快的数据库系统。你可以在MySQL网站http://www.mysql.com/why-mysql/benchmarks/上的性能比较主页上查到有关数据。
- **易使用**。MySQL是一种简单易用的高性能数据库系统，与其他大型数据库系统相比，MySQL的安装和管理工作要容易得多。
- **查询语言支持**。MySQL支持SQL语言，SQL是各种现代数据库系统的首选查询语言。
- **功能丰富**。MySQL是多线程的，允许多个客户同时与服务器建立连接。每个客户都可以同时打开并使用多个数据库。你可以通过好几种办法（如命令行客户程序、Web浏览器、GUI客户程序等）对MySQL数据库进行交互式访问，在输入查询命令后立刻看到查询结果。此外，MySQL还准备了C、Perl、Java、PHP、Python和Ruby等多种语言的编程接口。你还可以通过支持ODBC（Open Database Connectivity，数据库开放连接，一种由微软公司开发的数据库通信协议）功能和.NET的应用程序来访问MySQL数据库。也就是说，你既可以选用现成的客户程序来访问MySQL数据库，也可以根据具体的应用来编写相关软件。
- **优异的联网和安防性能**。MySQL是完全网络化的数据库系统，用户可以从因特网上的任意地

点去访问它，因此你完全可以把你的数据拿出来与任何地方的任何人共享。同时，MySQL还具备完善的访问控制机制，这就将那些不应该看到你数据的人拒之门外。此外，为了提供更进一步的安防措施，MySQL还支持使用SSL（Secure Socket Layer，安全套接字层）协议的加密连接。

- **可移植性**。MySQL既能够运行在多种版本的Unix和Linux操作系统上，也能够运行在Windows和NetWare系统上。MySQL可以运行在各种硬件设备上，包括高端服务器。
- **短小精悍**。与某些数据库系统巨大的硬盘空间消耗量相比，MySQL发行版本的硬盘占用量相对要小得多。
- **成本低廉**。MySQL是一个开源项目，只要遵守GNU组织的GPL（General Public License）许可证条款，就可以任意使用。这意味着MySQL在大多数情况下都是免费的。其次，如果是喜欢或需要正规安排或是不想接受GPL许可证约束的组织，还有商业许可证可供选择。
- **来源广泛**。MySQL很容易获得，只要你有Web浏览器，就能从许多地方下载它。如果你想知道某个组件的工作原理，对它的某个算法感到好奇，或者想进行安全检查，你完全可以通过源代码来钻研它。如果你不喜欢它的某个组件，也完全可以自行加以修改。如果你自认为发现了一个bug，可以报告给相关开发人员。

MySQL的技术支持怎么样？这个问题问得好，不能提供支持的数据库系统没什么用。本书就是一种支持，希望它可以满足你在数据库方面的需要（本书既然是第4版了，就表明它能做到这一点）。你还可以利用其他一些MySQL的相关资源。

- MySQL的发行版本都带有《MySQL参考手册》（*MySQL Reference Manual*），有在线版和印刷版。MySQL用户对这本手册都给予了很高的评价。这一点非常重要，因为如果没有人知道如何使用，再好的软件产品也会贬值。
- 如果你想得到正规的培训或者专业的技术支持，可以报名参加Sun公司开设的培训课程，或者与该公司签订技术支持和跟踪服务合同。
- MySQL社区有一些非常活跃的邮件列表，任何人都能订阅。这些邮件列表有很多专家级的参与者，许多MySQL开发人员都是它的常客。作为提供技术支持的电子资源，它们被很多订阅者认为是物有所值。

MySQL大家庭（包括开发人员和普通用户在内）是一个团结互助的群体。贴在邮件列表上的求助帖子通常在几分钟内就会得到回复。如果有人报告说发现了一个bug并得到确认，开发人员就会马上发布一个修补方案并经由因特网迅速传遍整个社区。与此形成鲜明对照的是，某些大厂商提供的技术支持服务令人困惑，那种不得其门而入的感觉实在让人着急上火。你遇到过这样的情况吗？我遇到过。

如果你正打算挑选一种数据库产品，那么MySQL绝对是理想的候选。使用MySQL既无风险，也不需要花费金钱。如果你遇到了问题，还可以通过邮件列表寻求帮助。当然了，做这样的评估必定会花费一些时间，但无论你原来计划使用哪种数据库产品，反正都要花时间评估。与很多其他的数据库产品相比，安装和测试MySQL的时间肯定少得多。

如果已经在运行其他RDBMS，该怎么办

如果你已经在使用某种数据库系统，但又颇受限制，那就绝对应该给MySQL一个机会。也许你觉得自己现有系统的性能不太好，也许它是一个专有产品而你又不想吊死在一棵树上，也许你想更换现

有的硬件设备而现有的软件系统却不支持，也许你现有的软件都是二进制代码而你更希望得到一种能够提供源代码的系统，也许你只是嫌它花钱太多……这一切都是你应该给MySQL一个机会的理由。你可以先通过本书熟悉一下MySQL的功能，再到MySQL邮件列表上提几个问题，然后再根据具体情况慎重抉择。

要明确的是，尽管所有主要的数据库引擎都支持SQL语言，但每种引擎支持不同的“方言”。请查阅本书关于MySQL所支持的SQL“方言”以及相关数据类型的章节。你也许会发现它们与你目前使用的RDBMS所支持的SQL版本区别太大，因而需要付出巨大的努力才能把你的应用程序迁移到MySQL系统上来。

当然，作为评估工作的一部分，应该先通过几个例子看看效果。这会让你在评估时获得宝贵的实际体验。MySQL的研发人员一直在努力让MySQL符合SQL语言标准，其效果之一就是让数据库应用程序迁移道路上的障碍随着时间的推移而不断减少，所以你的迁移工作很可能会比预期的容易许多。

MySQL提供的软件工具

MySQL的发行版本都附带以下几种工具程序。

- **一个SQL服务器**。运转整个MySQL的引擎，对MySQL数据库的访问和操作都要通过它才能实现。
- **客户程序和工具程序**。其中包括一个供你直接提交查询并查看其结果的交互式客户程序以及几个用来对数据库站点进行管理和维护的工具程序。有一个工具程序用来监控MySQL服务器，另外几个工具程序则负责数据的导入、备份、数据表问题检查等。
- **一个供你自行开发应用程序的客户端库**。这个函数库是用C语言写的，所以你可以用C语言来编写客户程序。此外，这个函数库还提供了一些供其他语言（如Perl、PHP和Ruby）使用的第三方接口。

除MySQL本身提供的软件外，有很多聪明人也使用MySQL编写一些小程序来提高工作效率，并把自己的成果拿出来与大家分享。在这些第三方工具软件里，有些能帮助你更加得心应手地使用MySQL，还有一些把MySQL的功能进一步扩展到Web站点建设等方面中。

本书能让你学到哪些东西

通过阅读本书，可以高效地掌握MySQL的使用方法，从而高效地完成自己的工作。你将会学到怎样把信息资料录入数据库，怎样构造出查询语句以迅速获得有关问题的答案。

即使不是程序员，也可以学习和使用SQL。本书内容的重点之一就是介绍SQL的工作原理。但熟悉SQL的语法并不代表你掌握了SQL的使用技巧，所以本书的另一个重点就是介绍MySQL的独特功能及其用法。

你将学习如何把MySQL与其他软件工具结合起来。本书还将介绍如何通过MySQL与Perl或PHP语言来为数据库的查询结果生成动态Web页面，以及如何自行编写MySQL数据库访问程序。自行编写的程序会大大拓展MySQL的功能，满足应用项目的具体要求。

对于那些负责MySQL安装的人员，本书将为他们介绍有关职责及具体工作流程。你将学会如何建立用户账户，如何备份数据库，以及如何保证数据库的安全。

本书各章内容

本书内容分四部分。第一部分集中讨论数据库应用方面的概念。第二部分的重点是如何使用MySQL编写你们自己的程序。第三部分的目标读者是数据库管理员。第四部分是几个参考附录。

第一部分：MySQL基础知识

第1章主要包括MySQL的用途与用法、交互式MySQL客户程序的使用方法、SQL基础知识和MySQL的常用功能。

如今，各种主流的RDBMS都能识别和理解SQL语言，但各种数据库引擎所使用的SQL语言彼此有着细微的差异。第2章重点介绍使MySQL有别于其他数据库系统的特色功能。

第3章主要包括MySQL为存储信息而提供的数据类型、各种类型的特点和局限性、它们的使用时机和使用方法，以及如何在相似的数据类型中作出选择，还有表达式的求值办法和各类型之间的转换机制等。

第4章讨论如何编写和使用存储在服务器端的SQL程序，包括各种存储函数、存储过程、触发器和事件。

第5章讨论如何使查询有效地运行。

第二部分：MySQL的编程接口

第6章介绍MySQL提供的几种API（Application Programming Interface，应用程序编程接口）以及本书所涉及的几种API之间的详细比较。

第7章讲述如何利用MySQL的C客户端库所提供的API来编写C语言程序。

第8章探讨如何利用DBI模块编写Perl脚本，包括独立的命令行脚本和用于网站的CGI脚本。

第9章介绍如何利用PHP脚本语言和PHP数据对象（PDO）的数据库访问扩展来编写用来访问MySQL数据库的动态Web页面。

第三部分：MySQL的系统管理

第10章介绍数据库管理员的工作职责，以及如何让数据库站点成功运行。

第11章详细介绍MySQL数据子目录（即MySQL用来存放各种数据库文件、日志文件和状态文件的地方）的组织布局和内容。

第12章阐述如何在操作系统开启和关闭时正确完成MySQL服务器的开启和关闭，如何在MySQL系统里建立用户账户，如何维护日志文件，如何配置存储引擎，如何优化数据库服务器，以及如何运行多个服务器，等等。

第13章介绍如何提高MySQL的安防水平以抵御各种入侵和破坏（可能来自数据库服务器主机的其他用户和网络客户端），如何配置你的MySQL服务器以支持SSL上的安全连接。

第14章阐释如何通过预防性措施来降低灾难的发生几率，如何备份数据库，如何在灾难真的发生时（即使采取了预防性措施）尽快恢复系统的运转。

第四部分：附录

附录A介绍如何获得并安装本书所提到的主要工具和示例数据库文件。

附录B详细说明MySQL数据类型。

附录C探讨在SQL语句中用来编写表达式的操作符和函数。

附录D介绍 MySQL服务器维护的各个变量和SQL语句变量的用法。

附录E描述MySQL支持的每个SQL语句。

附录F介绍MySQL发行版本所提供的程序。

说明　附录G、H、I需要上网获取。先访问www.informit.com/title/9780672329388，注册后可获取它们。也可以访问www.kitebird.com/mysql-book来获取它们[①]。

附录G介绍MySQL C客户端库所提供的数据类型和函数。

附录H讨论Perl DBI模块提供的方法和属性。

附录I介绍PDO扩展为在PHP中支持MySQL而提供的方法。

如何阅读本书

阅读本书的任何地方时，都应该同时尝试示例。这意味着你一定要先在计算机上安装MySQL，再安装示例数据库sampdb的有关文件，本书的许多示例都要用到sampdb数据库。获得和安装有关组件的办法与步骤可以在附录A里查到。

如果你是一位MySQL数据库系统或SQL语言的新手，请从本书的第1章开始学习。第1章介绍了MySQL与SQL的基本概念和使用入门，对加快本书后续章节的学习有很大帮助。然后再前进到第2章、第3章和第4章去学习如何描述和使用你自己的数据。这样，你就能有针对性地探索各种MySQL功能了。

即使你已经具备了一些SQL知识，也应该从第2章和第3章入手。不同的RDBMS系统所实现的SQL功能也不同，你应该首先弄清楚MySQL与你所熟悉的其他RDBMS系统有何区别。

如果你已经有了一些MySQL方面的经验但还需要进一步了解某些特定操作的原理，请把本书当做一本参考大全并根据需要有选择地查阅。你将发现书后的各个附录非常有价值。

如果你想编写能访问MySQL数据库的程序，请从第6章开始去学习有关API的章节。如果你想为自己的数据库开发一些便于使用的基于Web的前端访问程序，或者想为自己的数据库网站开发一些后端程序来增添动态内容，请阅读第8章和第9章。

如果要对MySQL和自己正在使用的RDBMS进行比较评估，本书的几个部分将有所帮助。如果想了解MySQL与你现有的SQL系统有何异同，请阅读本书第一部分中专门讨论数据类型和SQL语法的章节；如果你打算自己开发应用程序，请阅读第二部分中讨论编程的章节；如果你想了解MySQL需要何种级别的数据库管理功能，请阅读第三部分中有关管理的章节。如果你现在还没使用数据库，但正在对MySQL和其他数据库系统进行比较以作出选择，这些内容对你也将有很大的帮助。

书中涉及的软件及其版本

本书的第1版主要围绕MySQL 3.22版展开讨论并简要地介绍MySQL 3.23版。第2版把讨论范围扩大到了MySQL 4.0系列和MySQL 4.1系列的第一个发行版本。第3版讨论MySQL 4.1和MySQL 5.0中最早的几个发行版本。

本书是第4版，讨论的是MySQL 5.0。具体而言，本书将讨论MySQL 5.0和5.1版，以及MySQL 6.0

① 相应的中文译稿可在图灵网站（www.turingbook.com）本书主页上免费注册下载。——编者注

中最早发行的几个版本。本书的大部分内容仍适用于5.0和更早的版本，但我们不会特别指出特定于老版本的地方。

MySQL 5.0系列已经达到了通用阶段（即所谓的GA版），也就是说它已被认为能够稳定地运行在日常生产环境里。因为在MySQL 5.0系列的早期发行版本里有大量的修改，所以建议大家尽量选择最新的版本。在我编写本书的时候，5.0系列的最新版本是5.0.64。MySQL 5.1系列现处于备选版开发（Candidate Development）阶段，应该很快就会达到通用阶段。如果你想试试诸如事件调度器或XML支持之类的功能，你将需要MySQL 5.1。

如果你正在使用的MySQL版本早于5.0，本书讨论的以下几项功能将不可用。

- MySQL 5.0中增加的存储函数和过程、视图、触发器、脚本输入处理、真正的`VARCHAR`类型以及`INFORMATION_SCHEMA`。
- MySQL 5.1增加的事件调度器、分区、日志数据表和XML支持。

如果需要了解老版本，请访问MySQL官方文档网站http://dev.mysql.com/doc/，在那里可以查到每个版本的《参考手册》。

请注意以下几个没在本书里讨论的主题。

- 一些MySQL Connector组件，用户可通过它们访问Java、ODBC和.NET程序。
- NDB存储引擎和MySQL Cluster组件，它们用来提供以内存为介质的存储机制、高可用性和冗余。细节问题请查阅《MySQL参考手册》。
- 诸如MySQL Administrator和MySQL Query Brower之类的GUI（Graphical User Interface，图形化用户界面）工具。这些工具有助于在窗口环境里使用MySQL。

如果需要下载这些产品或查阅它们的文档，请访问http://www.mysql.com/products/或http：//dev.mysql.com/doc/。

至于书中涉及的其他一些主要软件，目前比较常见的版本都应该可以满足书中示例的需要。（请注意：PDO数据库访问扩展必须使用PHP 5或更高版本，而在PHP 4环境下无法工作。）各主要软件的最新版本如下所示：

软件包	版本
Perl DBI模块	1.601
Perl DBD::mysql模块	4.007
PHP	5.2.6
Apache	2.0.63/2.2.8
CGI.pm	3.29

书中提到的所有软件都可以在因特网上找到。附录A介绍如何获得并在自己的系统上安装MySQL、Perl DBI、PHP和PDO、Apache、CGI.pm等软件，如何获得本书通篇使用的`sampdb`示例数据库（其中包含本书讲述程序设计时会用到的示例程序）。

如果读者使用的是Windows，我将假设你有Windows 2000、XP、2003或Vista之类相对较新的版本。本书里讨论的某些功能，例如命名管道和Windows服务，较早的版本（Windows 95、98或Me）不支持。

排版约定

本书的排版要求如下所示。

- 文件名和命令等都用`Courier`字体表示。

- 命令中需要由读者输入的部分用**Courier加粗**表示。
- 命令中需要由读者替换为自己选择的内容的部分用*Courier斜体字*表示。

在需要进行交互操作的例子里，我将假设你会把命令输入到终端窗口或控制台窗口。为反映出上下文环境，我将通过命令行提示符来表明所运行的命令的执行环境。比如说，对于通过mysql客户程序输入的SQL语句，相应的命令行提示符将是mysql>。对于通过命令解释器输入的命令，提示符将是%，这个提示符表示命令可以在Unix系统或者Windows系统下使用，但你们看到的提示符到底是什么要取决于命令解释器。（对Unix用户而言，命令解释器就是你的登录shell；对Windows用户而言，它是cmd.exe或command.exe程序）。#提示符的意义比较特殊，它表示命令通过su或sudo命令由Unix系统上的root用户执行，而C:\>提示符则表示Windows系统上的专用命令。

下面的例子给出了一条应该从命令解释器输入的命令。%是提示符，不需要输入。为了输入这条命令，需要依次输入粗体字字符并用你自己的用户名替换斜体字：

```
% mysql --user=user-name sampdb
```

在SQL语句里，SQL关键字和函数名都用大写英文字母，而数据库、数据表、数据列的名称则全部用小写字母。

在语法描述中，方括号[]表示内容可选，可选的内容以垂直线字符|分隔。方括号内的列表是可选的，其具体内容应该是该列表里的某一个数据项。花括号{}内的列表是必不可少的，必须从该列表里选择一个数据项。

其他资源

如果你没能在本书里找到问题的答案，该怎么办呢？以下是一些软件的网站。

软件包	官方Web站点
MySQL	http: //dev.mysql.com/doc/
Perl DBI	http: //dbi.perl.org/
PHP	http: //www.php.net /
Apache	http: //httpd.apache.org/
CGI.pm	http: //search.cpan.org/dist/CGI.pm/

这些网站提供的信息资源有参考手册、常见问题答疑文档（Frequently Asked-Question，FAQ）和各种邮件列表等。

- **参考手册**。MySQL发行版本中自带的主要文档。这些文档的格式有很多种，网上还有它们的在线版本和可下载版本。

  PHP的使用手册也有好几种格式。
- **手册页面**。DBI模块及其MySQL专用驱动程序DBD::mysql的文档可以从命令行使用perldoc命令查阅。试试perldoc DBI和perldoc DBD::mysql命令。DBI的文档侧重于基本概念，而其MySQL专用驱动程序的文档则侧重于与MySQL有关的各种具体功能。
- **FAQ文档**。DBI、PHP、Apache各有各的FAQ文档。
- **邮件列表**。本书所涉及的一些软件有它们各自的邮件列表。如果你打算使用某个工具软件，那最好订阅一份与之有关的邮件列表。使用邮件列表上的归档文件也是个好主意。如果你不熟悉某个软件工具，你的很多问题就可能是很多前人已经问过（并得到回答）无数次的了；

你不必再提出类似的问题，因为它们的答案几乎都能在邮件列表的归档文件中搜索到。

不同的邮件列表有不同的订阅方式，下面这些URL地址可以为你提供相应的帮助。

软件包	邮件列表订阅站点
MySQL	http: //lists.mysql.com /
Perl DBI	http: //dbi.perl.org /support/
PHP	http: //www.php.net /mailing-lists.php
Apache	http: //httpd.apache.org /lists.html

- **其他网站**。除官方网站外，书中涉及的某些软件工具还另有一些提供其他信息（如示例程序的源代码和热门文章）的网站。这些网站大都可以通过官方网站上的链接找到。

目　录

第三部分 MySQL 的系统管理

第四部分 附 录

Part 1

第一部分

MySQL 基础知识

本部分内容

第 1 章

MySQL 和 SQL 入门

本章的主要内容是 MySQL 关系数据库管理系统和 MySQL 所使用的 SQL（Structured Query Language，结构化查询语言）的基础知识。本章将介绍大家应该掌握的术语和基本概念，描述书中示例所用到的 sampdb 示例数据库和 MySQL 数据库的创建与交互操作。

如果你在数据库系统方面是一个新手，不能肯定自己是否需要这种东西，就应该从本章开始学习。如果你完全不了解 MySQL 或 SQL，也应该从作为入门指南的本章入手。已经具备一定的 MySQL 或其他数据库系统使用经验的读者可以跳过本章内容。但我希望大家至少要看看 1.2 节，熟悉一下 sampdb 数据库的用途与内容，因为我们将在全书的示例中反复用到它。

1.1 MySQL 的用途

本节将描述 MySQL 数据库系统的用武之地，让大家了解 MySQL 能用来做哪些事情以及它们会对你的工作有什么样的促进和帮助作用。如果你用不着这些描述就已经信服了数据库的作用——也许你心里正有一个难题，急于让 MySQL 运转起来以解决它——不妨立刻前进到 1.2 节。

从本质上讲，数据库系统只不过是一套对大量信息进行管理的高效办法而已。信息有各种来源。例如，信息可以是科研数据、商业账目记录、客户定单、体育比赛成绩、销售业绩报告、个人爱好资料、人事档案记录、bug 报告、学生考试成绩，等等。虽然数据库系统能处理各种各样的信息，但单纯因为其本身而安装并使用它却不见得有必要。假如某项工作已经有了一个很好的解决方案，而你却在“为使用而使用”的心理驱使下引入了一个数据库系统，那就太不明智了。商品采购清单就是一个很好的例子：在出发前，先把想买的东西列在一张纸上；到商店后，每买到一样东西，就把它从清单上划掉；等采购完毕时，这张纸就可以丢掉不要了。几乎没有人会因为这种事情去动用数据库。如果你有掌上电脑，你应该会用掌上电脑的记事本功能来处理商品采购清单，而不会选用数据库。

当需要组织和管理的信息很多或者信息本身很复杂时，仍手工处理数据记录就会变得力不从心，而使用数据库系统则会让这些事变得轻而易举。对那些每天要处理上百万项交易的大公司来说，数据库不可或缺。但即便是某位用户出于个人爱好而收集整理信息这种小规模操作，也可能需要一个数据库。数据库可以帮上大忙的场景并不难想象，因为即便是很少的信息也很难管理。请考虑以下情况。

- 你开了一家木工厂并雇了几位员工。你需要维护一份员工名单和一份工资表，得把自己在何时给哪些员工发过工资的事记下来，还得把工资总数加起来好向政府有关部门报税。你还需要记录工厂接过的每一单木工活，以及每单木工活都由哪几位员工负责完成等事情。

- 你开了一家汽车配件连锁店。为了完成顾客的订单，你需要随时了解某个零件在哪一家分店里还有库存。
- 你在长年的科研工作中积累了大量的数据。为了发表研究成果，你必须对这些数据进行筛选和分析。这是一个沙里淘金般的工作，需要从大量原始数据生成汇总信息，提取典型的数据来进行统计分析，再根据分析结果推导出结论来。
- 你是一名教师，需要记录学生的考试成绩和出勤情况。每进行一次考试或测验，就把学生们的成绩记录下来。把成绩记到成绩本上并不复杂，但今后想分析这些成绩时可就麻烦了。对各次考试的成绩进行排序以确定分数线，学期结束时为每位学生计算总评成绩，统计学生们的出勤情况，等等，你一定不会手工完成这些工作。
- 你在某机构担任秘书一职，负责维护该机构的成员名录。（这个机构可能是专业团体、俱乐部、交响乐团、健身俱乐部等。）你每年都要为大家打印一份成员名录，每当有成员资料发生变化，就得用字处理软件修改这份文档。文档使你的很多好想法都无法实现，所以你对目前的状况感到很厌倦。很难对成员名录进行多种排序，很难从中选出指定的部分（例如只列出人名和电话号码），很难把符合某种条件的成员（如需要延续成员资格的人）都找出来（所以你每个月都得把这些成员一个不落地找出来并给他们寄去续会通知）。你听说过“无纸办公”，知道它指的是电子化的办公形式，但你并不了解它对你有什么好处。成员名录已经是电子化的了，可具有讽刺意味的是，除了把名录打印成册以外，任何其他类型的工作都不容易完成。

在上面列举的这些场景里，有的信息量很大，有的信息量很小。但它们有一个共同的特点，即这些工作原先都是手工完成的，但引入一个数据库系统将大幅提高工作效率。

那么，诸如 MySQL 之类的数据库系统会给你带来哪些特别的好处呢？这要看你的具体需要和目标到底是什么。在上面那些例子里，不同的场景有着不同的要求。下面，我们将以一个常见的情况为例来说明数据库的作用。数据库管理系统通常被人们用来取代文件柜，而事实上，数据库系统也像一个巨大的文件柜，只是它里面已经有很多预先建立好的存档功能。与手工方式相比，以电子化手段来管理信息的优势非常明显，同时也非常重要。我们来看一个例子，假设你要为牙科诊所管理顾客资料。下面是一些 MySQL 的存档功能，可以为你带来的巨大帮助。

缩短信息记录的录入时间。当需要添加一项新的信息记录时，你用不着拉开文件柜的各个抽屉以确定需要把这条记录添加到什么地方。你只需把这条记录提交给存档系统，让它把该记录存放到适当位置。

缩短信息记录的检索时间。当需要查找某条信息记录时，你用不着亲自拉开文件柜的各个抽屉就能找到你想要的资料。如果你想给最近没来参加定期检查的人们发一封提醒信，就完全可以让存档系统去把这些人的资料查找出来。当然，这与你让另一位员工“把最近 6 个月没来参加定期检查的人查出来”的情况是不同的。如果你有一个数据库，就可以直接用下面这条看起来很奇怪的语句完成这项工作：

```
SELECT last_name, first_name, last_visit FROM patient
WHERE last_visit < DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

如果你从没见过类似的东西，这条语句看起来会吓人。以前需要花费一小时的时间才能得到结果，而现在你只需一两秒钟就能完成，这一点还是相当吸引人的。（现在，请不要被这条奇怪的语句吓倒。用不了多长时间，你就不再会对它感到陌生了。事实上，等你学完本章内容，就会明白这条语句到底有什么含义了。）

灵活的信息检索顺序。你用不着按照当初存储记录的顺序（例如按患者的姓氏顺序）来检索它们。你可以让你的存档系统按你希望的任意顺序来提取信息：按姓氏也行，按医疗保险公司名称的顺序也行，按最近一次就诊时间的顺序也行，等等。

灵活的输出格式。找到想要的资料后，你不必再把它们手工抄写下来。你可以让存档系统把它们生成为一份名单。有时候，你需要把信息打印出来；有时候，你可能想把它们用在另一个程序里。（例如，在生成一份最近没来参加定期诊断的患者名单后，你可以把这些资料送到一个文字处理软件里去打印催诊通知，然后再寄给那些患者。）也许你只对汇总信息（如总共有多少人没来参加定期诊断）感兴趣。有了数据库，你就用不着再亲自统计这些人数了，存档系统会轻而易举地为你生成汇总信息来告诉你。

信息记录能同时被多名员工使用。在“有纸办公”的年代，如果有两个人同时需要查看同一份资料，第二个人就必须等第一个人把资料放回原处之后才能拿到它。有了 MySQL，你就能让多位员工同时使用同一份资料了。

信息记录的远程访问和电子传输。“有纸办公”只允许你在信息资料的存放地点使用它们，或者让别人给你复印一份送过来。电子信息记录则允许对这些记录进行远程访问或者电子传输。如果你的牙科诊所设有分支机构，分支机构里的医护人员就能从他们自己的办公地点存取资料了。你不再需要通过信使来传送这些资料。如果别人想获得记录却又没有与你一样的数据库软件，那么，只要他能使用电子邮件，你就可以把他想要的资料找出来并通过电子手段传送给他。

如果你曾经使用过数据库管理系统，就会亲身体会到上面列举的种种好处，而你现在的想法可能已经超越“电子文件柜”的范畴了。很多企事业单位现在都把他们的数据库与网站结合起来使用，这是一个很好的例子。现在，假设你所在的公司有一个库存商品数据库，每当有顾客打电话来询问仓库里是否存有某种货物及其价格时，服务台员工就会用到这个数据库。这只是数据库比较传统的用法。如果你所在的公司还有一个可供顾客访问的网站，就可以增加一项新的服务项目——为顾客提供一个查询页面，让他们自己查询商品库存情况和价格信息。这样，顾客就能迅速地获得他们想要的资料，这些资料是从你的数据库里查出来的，而你为他们提供的库存商品查询功能又是自动完成的。顾客很快就能获得资料，不必再拿着电话筒听占线忙音，也不必再受你们公司上下班时间的限制。而每一位使用你们公司网络的顾客都替你省下了一小笔需要支付给服务台员工的工资。（光是如此节省下来的钱恐怕就能抵消网站本身的开支了。）

你还可以更进一步地发挥数据库的作用。基于 Web 的库存查询功能不仅能满足顾客的需求，对你的公司也有莫大的好处。这些查询能让你了解顾客想买哪些商品，而查询结果则能让你了解你是否能够满足顾客的要求。如果你的仓库里没有顾客想要的东西，你可能会失去这笔生意。所以把来自顾客的库存查询信息（他们想买什么，你是否有足够的库存等）记录下来不失为一个好主意。你可以根据这些信息来调整库存，向顾客提供更好的服务。

说了半天，MySQL 到底是怎样工作的呢？寻找这一答案的最佳办法是亲身体验一下。为此，我们需要先建立一个示例数据库。

1.2　示例数据库

本节描述本书后续各章中使用的两个示例数据库。在你学习 MySQL 使用方法的过程中，这两个数据库将充当各示例的信息源。这两个示例数据库是我们根据此前介绍的以下两种场景生成的。

- 你担任某机构秘书一职。这个机构有一些特点。它是由一些对美国历史很感兴趣的人自发组织起来的，我们不妨把它称为美国历史研究会。出于个人的爱好，那些会员将自愿定期交纳一定的会费以维持其会员身份。会员交纳上来的会费将用于支付研究会的开支，如会员刊物*Chronicles of U.S. Past*（美国历史年表）的印刷费用。这个研究会建有一个小规模的网站，但还没有得到充分的开发和利用，你可能想改变它。
- 考试分数记录。你是一名教师，在学期中负责考试与测验，记录考试分数，给学生打分。在学期末，你对学生的成绩作出总评，并把总评成绩和出勤情况上报给学校办公室。

下面，我们进一步分析这两种场景的需求。

- 你需要决定想从数据库得到哪些东西——也就是你想达到的目标。
- 你需要决定想把哪些东西放到数据库里——也就是你想记录的信息。

“你想从数据库得到哪些东西”位于“你想把哪些东西放到数据库里”的前面，这看起来似乎有些本末倒置，因为无论如何，你得先把数据放到数据库里才能对它们进行检索。事情是这样的，如何使用数据库主要取决于你想达到的目标，而目标又主要取决于你将从数据库检索出来的东西而不是你放到数据库里的东西。如果你今后根本不会使用你放入数据库里的信息，那就用不着浪费时间和精力把它们放进去了。

1.2.1 “美国历史研究会”场景

在这个场景里，你担任着这个研究会的秘书职务。眼下，你正使用一份文档维护这个研究会的会员名录。当然，利用文档来打印会员名录还是很容易完成的，可要是还想利用它再做些别的事情就没那么容易了。你有以下几个目标。

- 你希望能够根据不同情况把会员名录以其他格式输出，只输出符合特定要求的资料。首先，每年生成一份全体会员的名录——这是研究会的传统工作之一。你还想利用会员名录做一些其他工作，例如向研究会提供一份最新的会员名单以便邀请他们出席研究会的周年宴会。这两项工作需要用到的信息是不同的：全体会员名录需要用到每位会员的完整资料，而周年宴会只需用到会员的姓名（这项工作可不容易用文档来完成）。
- 你希望能够根据各种限制条件有选择地找出部分会员来。比如说，你想知道近期有哪些会员需要续交年费以保留其会员资格。另外，你还希望能够根据一些关键词把会员们分类，这些关键词代表每位会员对美国历史上的不同时期的兴趣，比如“Civil War”（南北战争）、“Depression”（经济大萧条）、“civil right”（民权法案）、“Thomas Jefferson”（托马斯·杰弗逊）总统的生平事迹，等等。会员有时希望你能为他们提供一份与自己志趣相投的其他会员的名单，而你也非常想满足这些会员的愿望。
- 你还想把研究会的会员名录发布到研究会的网站上去。这对会员和你都有好处。如果你能通过某种自动化的过程把会员名录转换为Web页面，在线版本的会员名录将总是最新的，这是纸张形式无法达到的。如果这份在线名录还支持检索功能，会员们就能轻松地自行查找他们感兴趣的信息了。比如说，如果某位会员想知道谁还对南北战争感兴趣，他完全可以自己去查询，用不着等你去帮他查找，而你也能抽出时间去做些别的事情。

我得承认，数据库并不是世界上最令人激动的东西，所以我也不打算鼓吹说使用数据库能够激发人们的创造性思维。不过，如果你不把信息看成是要解决的难题（和你看待文档一样），而是可以轻

松操作的东西（你希望使用 MySQL 时是这样），就肯定会发现很多使用这些信息的新方法，如下所示。

- 如果数据库中的信息能够以在线名录的形式添加到网站上，这些信息流肯定会发挥出其他的作用。比如说，如果会员能够以在线方式修改本人的资料来刷新数据库，就用不着再由你来负责录入工作了，会员资料也更准确。你一定不想自己做这些修改工作，而研究会也没有预算再雇一个人。
- 如果你把电子邮件地址也添加到数据库里，就可以通过电子邮件来提醒会员注意更新自己的资料。在发给会员的邮件里，你可以附上他们的现有资料，让他们自己去核查这些信息并通过研究会网站提供的有关功能进行必要的修改。
- 数据库将大大拓展研究会网站的用途，并不仅仅局限于会员名录。研究会有自己的会刊 *Chronicles of U.S. Past*，每期会刊里都有一个专为儿童准备的历史知识测验栏目。假设最近几期会刊里的小测验题目都集中在美国总统的生平事迹方面。你在研究会的网站上也可以开设一个类似的儿童栏目并以在线方式给出历史知识小测验题目。这个栏目甚至可以办成互动形式——小测验的题目都由Web服务器以实时方式直接从数据库里提取出来并展示给站点访客。

太让人兴奋了！你脑子里想到的这些数据库应用让你有点忘乎所以了。在重新回到现实之后，你要开始考虑以下一些实际的问题。

- 这是不是有点野心勃勃？实现起来工作量是否太大？

  要知道，空想总是要比实干来得容易，我也不想虚伪地告诉大家说这些设想都很容易实现。不过，等你学习完这本书的时候，我们现在勾勒出来的这些设想将全部成为现实。请记住这样一个道理：一蹴而就是不可能的。我们将把这些工作分解，然后逐一实现。
- MySQL能够胜任所有这些工作吗？

  不，它不能，至少单靠它自己不能。例如，MySQL 不具备直接开发 Web 程序的功能，你必须把它与其他软件开发工具结合起来才能完善和扩展它的功能。

  我们将利用 Perl 脚本语言和 Perl 语言中的 DBI（Database Interface，数据库接口）模块来编写用来访问 MySQL 数据库的脚本程序。Perl 有着强大的文本处理功能，能够对数据库的查询结果进行极其灵活的处理并生成各式各样的输出。例如，我们可以用 Perl 生成一份 RTF（Rich Text Format，富文本格式）的会员名录，而任何一种字处理软件都能识别出这种格式，也可以使用用于 Web 浏览器的 HTML 格式。

  我们还将用到另一种脚本语言，PHP。PHP 特别适合用来编写 Web 应用，同时它也很容易与数据库合作。这将使你能够从 Web 页面来开始 MySQL 查询，再把数据库的查询结果包含在一个新生成的页面里。PHP 与一些 Web 服务器软件（包括世界上最流行的 Web 服务器软件 Apache）配合得非常好，这使得提供查询页面和显示查询结果很容易。

  MySQL 能够非常好地与这些工具集成在一起，你可以灵活组合这些开发工具以达到你心中的目标。有些套装开发工具声称自己有着极高的“集成度”，被宣传得几乎是无所不能，可实际上却只能实现套装组件间的配合，与其他组件的配合并不理想。大家千万不要被蒙蔽。
- 最后，也是最重要的一个问题：这要花费多少钱？别忘了，美国历史研究会并没有多少预算。也许有点让人难以置信，可采用上述组合的解决方案的确没有什么成本。如果你有一些关于数据库系统方面的常识，就该知道它们通常都很昂贵。与此形成鲜明对比的是，MySQL 几乎总是免费的。即便是在需要技术支持服务和维护作业的企业级环境里，使用 MySQL 作为数据库系统的成本也是相对低廉的。（详见 www.mysql.com 网站。）我们将使用的其他工具（Perl、

DBI、PHP、Apache）都是免费的，因此，如果把所有事情都考虑进来的话，你花很小的成本就可以组建一个实用的系统。

你可以任意选择操作系统来开发你的数据库。我们将介绍的软件开发工具全都能在 UNIX（包括 BSD UNIX、Linux、Mac OS X 等）和 Windows 操作系统上使用，但专门针对 UNIX 或 Winows 的 shell 脚本或批处理脚本例外。

1.2.2　考试记分项目

现在我们再去看看另一场景中使用的示例数据库吧。在这个场景里，你是一位负责记录学生考试成绩的教师。你想把手工记录的学生成绩簿转换到一个使用 MySQL 的电子系统上去。在这个场景里，从数据库里获取信息的方式与你目前使用学生成绩簿时差不多，如下所示。

- 每次测验或考试之后，都要把考生的成绩记录下来。如果是考试，你还需要对分数进行排序以评定级别（A、B、C、D、F）分数线。
- 在学期结束时，把学生这一学期的总分数计算出来，对这些总分数进行排序，由此评定级别。在计算总分数的时候，你可能需要进行加权计算以区别考试成绩和测验成绩，因为考试通常要比测验重要。
- 在学期结束的时候，还要向学校办公室提交学生的考勤情况。

你的目标是不再以人工方式对学生们的考试分数和出勤情况进行排序和汇总。换句话说，你希望每次考试后的分数排序工作，以及各学期末的学生总评分和出勤情况的统计工作都能够交给 MySQL 去完成。为此，你需要知道班级里的学生名单、他们每次考试或测验的分数，以及缺勤的学生姓名。

1.2.3　关于示例数据库的说明

如果你对我们安排的“美国历史研究会”或者“考试记分项目”没有什么兴趣，你可能想知道它们和你有什么关系。要知道，这些场景只是一些例子，它们本身并不是我们的学习目标。它们只是我们在学习使用 MySQL 及相关工具时需要用到的一种载体。只要稍微动点脑筋，你就能看出这些示例数据库上的查询对解决你本人真正关心的具体问题有帮助。我们不妨假设你在我前面提到的那个牙科诊所里工作。虽然你在这本书里看不到多少与牙科医学有关的查询，但书中的很多查询都能运用到患者资料或办公室资料的管理工作中。比如说，在“美国历史研究会”场景里，你需要把近期必须续费才能保持其会员资格的会员找出来；而这与你把近期需要来牙科诊所进行定期检查的患者找出来的情况是类似的——它们都是与日期有关的查询。因此，只要你学会了如何写出一个用来找出哪些会员需要续费的查询，就可以运用类似的原理写出一个用来找出哪些患者需要就诊的查询来。

1.3　数据库基本术语

你大概注意到，你已经看了好多页了，却至今仍未在这本讲数据库的书中遇到太多专业术语和技术词汇。事实上，到目前为止，虽然已经粗略描述过示例数据库的用法，但我仍没说过一句关于什么是数据库的话。可是，既然我们将要设计一个数据库并开始实现它，我们就无法继续回避有关的术语，它们正是本节要介绍的内容。本节将描述书中使用的一些术语，希望大家能够掌握它们的含义。值得庆幸的是，与关系数据库有关的概念大都比较简单。事实上，人们喜欢关系数据库的很大一部分原因就在于它们的基本概念都很简明易懂。

1.3.1　数据库的组织结构

在数据库的世界里，MySQL 被划分到关系数据库管理系统（Relational Database Management System，RDBMS）的范畴内。我们可以把这个短语划分为以下几个部分。

- 数据库（database，即RDBMS里的DB）就是一个用来存放信息的仓库，它们构造简单，遵守一定的规则：
  - 数据库里的数据集合都存放在数据表（table）里；
  - 数据表由数据行（row）和数据列（column）构成；
  - 一个数据行就是数据表里的一条记录（record）；
  - 记录可以包含多个信息项，数据表里的每一个数据列都对应一个信息项。
- 管理系统（management system，即RDBMS里的MS）指的是用来对数据进行插入、检索、修改、删除等操作的软件。
- 关系（relational，即RDBMS里的R）表示RDBMS是DBMS中的一种，这种DBMS的专长就是把分别存放在两个数据表里的信息联系（即相互匹配）起来，而这种联系是通过查找两个数据表的共同元素来实现的。RDBMS的威力在于它能方便地抽取出数据表里的数据并把它们与其他相关数据表的信息结合起来，为那些单独利用某个数据表无法找到答案的问题提供答案。（事实上，"关系"的正式定义与这里有所不同，为此我向纯粹主义者们表示道歉，但这里的定义有助于解释RDBMS的用途。）

关系数据库是如何把数据组织到数据表里的？又是如何把来自不同数据表的信息联系在一起的呢？我们来看一个例子。假设你有一个包含横幅广告服务的网站，并与一些想登广告的公司签订了合同。每当有访客点击了你的某个页面时，你就会把一条广告嵌入到页面里，将其发送给访客的浏览器。同时，你还会向刊登这条广告的公司收取一笔小小的费用。这样就是一次广告点击。为了记录这些信息，你使用了 3 个数据表（如图 1-1 所示）。company 数据表由以下几个数据列构成：公司名称（company_name）、公司编号（company_num）、地址（address）和电话号码（phone）。ad（广告）数据表由以下几个数据列构成：广告编号（ad_num）、拥有该广告的公司的编号（company_num）和该广告每被点击一次的计费标准（hit_fee）。hit（点击情况）数据表由以下几个数据列构成：广告编号（ad_num）和该广告被发送给浏览者的日期（date）。

有些问题只用一个数据表就能得到答案。比如说，如果你想知道有多少家公司与你签订了广告合同，你只要数一数 company 数据表总共有多少行就行了。如果你想了解在某个给定的时间段内有多少次广告点击，也只需用到 hit 数据表。可有些问题会比较复杂，需要查询多个数据表才能找出答案。例如，在 7 月 14 日这一天里，Pickles 公司的各条广告分别被点击了多少次？要想回答出这个问题，就必须把这 3 个数据表都用上，如下所示。

(1) 在 company 数据表里根据公司名称（Pickles, Inc.）查出对应的公司编号（14）。

(2) 利用这个公司编号在 ad 数据表里查找与之匹配的记录以确定相关的广告编号。我们找到了两个符合条件的广告，编号是 48 和 101。

(3) 利用这两个广告编号从 hit 数据表里把落在给定日期范围内的匹配记录找出来，再对匹配到的记录个数进行统计。最后，我们查出编号为 48 的广告有 3 个匹配，编号为 101 的广告有 2 个匹配。

company数据表

company_name	company_num	address	phone
Big deal, Ltd.	13	14 Grand Blvd.	875-2934
Pickles, Inc.	14	59 Cucumber Dr.	884-2472
Real Roofing Co.	17	928 Shingles Rd.	882-4173
GigaFred & Son	23	2572 Family Ave.	847-4738

ad 数据表

company_num	ad_num	hit_fee
14	48	0.01
23	49	0.02
17	52	0.01
13	55	0.03
23	62	0.02
23	63	0.01
23	64	0.02
13	77	0.03
23	99	0.03
14	101	0.01
13	102	0.01
17	119	0.02

hit数据表

ad_num	date
49	July 13
55	July 13
48	July 14
63	July 14
101	July 14
62	July 14
119	July 14
102	July 14
52	July 14
48	July 14
64	July 14
119	July 14
48	July 14
101	July 14
63	July 15
49	July 15
77	July 15
99	July 15

图 1-1 横幅广告数据表

听起来太复杂了，但这正是关系数据库系统所擅长的。而且，虽然看起来很复杂，但上面这几个步骤也只是几个简单的匹配操作而已：我们把一个数据表与另一个数据表联系起来，看前一个数据表的数据行取值是否与后一个数据表的数据行取值相匹配。把这几步简单的操作推广开来，我们就能找出各种问题的答案：各家公司分别有多少个不同的广告？哪家公司的广告最受欢迎？每个广告会带来多少收益？在本结算期内，每家公司应该支付你多少广告费？

现在，你对关系数据库理论的了解已经足以让你读懂本书后续章节的内容了，我也不想再用“第三范式”（Third Normal Form）、“实体联系图”（Entity Relationship Diagram）之类的枯燥概念去烦扰大家。（如果你想了解这些概念，我建议你去读读 C. J. Date 或 E. F. Codd 的著作。）

1.3.2 数据库查询语言

为了与 MySQL 交互，你需要使用一种名为 SQL（Structured Query Language，结构化查询语言）的语言。SQL 是今天的标准化数据库语言，在各种主流的数据库系统上都能使用（也有一些实现是供应商特有的）。SQL 中的各种语句使它能够高效率地与你的数据库进行互动。

与其他计算机语言一样，初次接触 SQL 的人往往会觉得它很奇怪。例如，在创建数据表的时候，你必须告诉 MySQL 这个数据表的结构是什么样的。很多人会把数据表想象成一个表格或者一幅图，但 MySQL 却不这么想，所以你在创建数据表时必须写出下面这样的代码：

```
CREATE TABLE company
(
  company_name CHAR(30),
  company_num  INT,
  address      CHAR(30),
  phone        CHAR(12)
);
```

这类语句往往会让 SQL 新手产生畏难情绪，不过请放心，不是程序员也能学会熟练运用 SQL。随着对这种语言的熟悉程度的加深，你对 `CREATE TABLE` 这类语句的看法也会悄悄地转变——它们不再是一团难以理清的乱麻，而是能帮助你描述信息的工具。

1.3.3 MySQL的体系结构

MySQL 采用的是客户/服务器体系结构。因此，当你使用 MySQL 的时候，你实际是在使用两个程序。第一个程序是 MySQL 服务器程序，指的是 mysqld 程序，它运行在存放着你的数据库的机器上。它负责在网络上监听并处理来自客户的服务请求，根据这些请求去访问数据库的内容，再把有关信息回传给客户。另一个程序是 MySQL 客户程序，它们负责连接到数据库服务器，并通过向服务器发出查询命令来告知它们需要哪些信息。

MySQL 的大多数发行版本包括数据库服务器和几个客户程序（在 Linux 下使用 RPM 包时，有单独 的服务器和客户 RPM 包，所以这两个都要安装）。你得根据自己的具体情况来选用一种客户程序。mysql 是最常用的客户程序，它是一个交互式的客户程序，你通过它发出查询命令并查看结果。mysqldump 和 mysqladmin 是两个主要用于数据库管理的客户程序，前者用来把数据表的内容导出到一个文件里，后者用来检查数据库服务器的工作状态和执行一些数据库管理方面的任务，例如通知数据库服务器停止运行等。MySQL 发行版本里往往还有其他的客户程序。MySQL 还提供了一个客户程序开发库，如果 MySQL 自带的标准客户程序不能满足你的应用要求，你就可以自行编写一些程序来解决问题。这个开发库可以从 C 语言程序里直接使用。如果你偏爱其他编程语言，还有其他语言（Perl、PHP、Python、Java、Ruby 等）的编程接口可供选用。

本书讨论的客户程序都是从命令行使用的。如果想试试使用 GUI（图形用户界面）并提供点击功能的工具，请访问 http://www.mysql.com/products/tools/。

MySQL 的“客户/服务器”体系结构有以下一些好处。

- 并发控制（concurrency control）由服务器提供，因而不会出现两个用户同时修改同一条记录的现象。来自客户的请求全都要经过服务器，由服务器来安排处理它们的先后顺序。即使出现多个客户同时请求访问同一个数据表的情况，也用不着由这些客户去发现对方并进行协商。它们只负责把请求发往服务器，而谁先谁后的事则完全由服务器去决定。

- 你不必非得在存放着你的数据库的那台机器上登录。MySQL 知道该如何在因特网上运行，所以你可以在任意地点运行 MySQL 客户程序，而这个客户程序能够通过网络寻找到服务器。地理距离根本不是问题，你可以从世界任何一个角落访问服务器。即使服务器位于澳大利亚而你带着一台笔记本电脑旅行到了冰岛，你也能访问到自己的数据库。可这是否意味着别人也能通过因特网看到你的数据呢？答案是“不”。MySQL 有一个灵活的安防系统，只有得到你授权的人才能访问你的数据。而且，你还可以进一步限制这些人只能做你允许他们做的事。比如说，财务部的 Sally 应该有查看和修改数据记录的权限，可服务部的 Phil 却只应该有查看它们的权限。总之，你可以把这种访问权限控制细化到每一个人。从另一方面讲，如果只想拥有一个完全属于你自己的系统，你也完全可以把访问权限设置成只允许客户程序从运行着服务器的那台主机上连接。

除原来以客户/服务器方式运行的 `mysqld` 服务器程序外，MySQL 又新增了一个库函数形式的服务器 libmysqld，你可以把它链接到程序里以编写出独立的基于 MySQL 的应用程序。因为它被嵌入在各个应用程序里，所以人们把 libmysqld 称为“嵌入式服务器库”（embedded server library）。嵌入式服务器与客户/服务器方案的主要区别是它不需要网络。这使我们能够方便地制作出这样一种“自给自足”的应用软件包来：它们对外部操作环境的要求更低，与数据库有关的操作完全由它自己来负责完成。这既是它的优点，也是它的局限性——如果机器里没有安装 MySQL 软件，其他软件包就无法访问该主机上的数据库了。

MySQL 与 mysql 的区别

为避免混淆，请注意，MySQL 指的是一个完整的 MySQL RDBMS，而 `mysql` 则是一个特定的客户程序的名字。它们的发音相同，但代表的却是不同的事物，所以本书要用大写和小写字母来区分它们。

说到发音，《MySQL 参考手册》给出的是“ my-ess-queue-ell ”。但 SQL 却有“ sequel ”和“ ess-queue-ell ”两种读法。

1.4 MySQL

好了，大家应该了解的预备知识也就是这么多。下面该让本书的主角 MySQL 出场了！

安排这样一节是为了帮助大家熟悉 MySQL 的基本用法。本节是这样安排的：首先创建示例数据库和几个必要的数据表，再通过示例数据库介绍如何对数据表里的信息进行插入、检索、删除、修改等操作。通过这些步骤，你将学到以下技能。

- MySQL能理解的SQL的基本知识。MySQL所使用的SQL语言与其他RDBMS使用的版本有着细微的差异。因此，即便你以前曾经接触过其他的RDBMS并有一定的SQL使用经验，也应该快速浏览一下本节的内容。
- 使用MySQL自带的标准客户程序与MySQL服务器通信。前面讲过，MySQL采用的是“客户/服务器”体系结构，服务器将运行在存放着数据库的主机上，而客户需要通过网络来连接到服务器上。本节的重点内容是`mysql`客户程序，它负责接收你发出的SQL查询命令，把它们发送到服务器执行，再把执行结果显示给你看。我们之所以会把`mysql`作为介绍重点，是因为它能在MySQL支持的任何一种平台上运行，是把你与数据库服务器直接联系在一起的纽带。本

节中的某些示例还用到了另外两个客户程序mysqlimport和mysqlshow。

我们给书中的示例数据库取名为 sampdb，但读者也许需要给它另外起个名字才行。比如说，也许在你的系统上已经有其他人把自己的数据库命名为 sampdb 了，或者你的 MySQL 管理员给这个示例数据库另外指定了一个名字。如果遇到这类情况，请把本书示例中的 sampdb 替换为你实际使用的示例数据库名称。

出现在本书示例中的数据表名称用不着任何改动就可使用，哪怕在你的系统里有多名用户都各自安装了一份示例数据库。在 MySQL 里，只要数据库的名字没有出现重复，数据库里的数据表是允许同名的。MySQL 将数据表限制在各自的数据库，防止互相干涉。

1.4.1　如何获得示例数据库

本节有时会用“sampdb 数据库发行版本”（或“sampdb 发行版本”，因为示例数据库的名字是 sampdb）来指称有关文件。这些文件里存放着用来安装示例数据库的查询命令和数据，它们的获取办法和安装步骤可以在附录 A 里查到。解压缩之后，安装过程将自动创建一个名为 sampdb 的子目录并把有关的文件存放到里面。顺便给大家提个建议：在拿 sampdb 数据库练手之前，最好先切换到这个子目录里。

如果想无论当前在哪个子目录都可以方便地运行 MySQL 程序，就应该把包含着那些程序的 MySQL bin 子目录添加到你的命令解释器的搜索路径里去。这很容易做到，按照本书附录 A 里给出的步骤把该子目录的路径名添加到你的 PATH 环境变量设置里去就可以了。

1.4.2　最低配置要求

要想试用本节中的示例，必须满足以下几项基本要求：

- 你的系统已经安装了MySQL软件；
- 你有一个用来连接数据库服务器的MySQL账户；
- 你有一个示例数据库。

MySQL 客户程序和 MySQL 服务器是必不可少的。MySQL 客户程序必须安装在你本人操作的机器里；服务器可以安装在你的机器里，也可以安装在别的主机里——只要你有对它的连接权限，MySQL 服务器可以放在任何地方。获得和安装 MySQL 的步骤请参考附录 A。如果你的网络连接需要经过一家 ISP（Internet Service Provider，因特网服务提供商），请提前查明该 ISP 提供的服务项目里有没有 MySQL。如果没有这个服务项目，并且该 ISP 也不准备安装它，请更换你的 ISP，选择提供 MySQL 的供应商。

除 MySQL 软件外，你还必须有一个 MySQL 账户。否则，你将无法连接 MySQL 服务器，也就无法创建你的示例数据库和其中的数据表。（如果你已经有了一个 MySQL 账户，可以直接拿过来用。但我建议你还是另外申请一个专供学习本书时使用的账户比较好。）

现在，我们遇到了一个“先有鸡，还是先有蛋”的问题：要想申请一个用来连接 MySQL 服务器的账户，你必须先连接到这个服务器上才行。一般来说，这需要在运行着 MySQL 服务器的主机上以 MySQL 的 root 用户身份登录，再用 CREATE USER 和 GRANT 语句新创建一个 MySQL 账户，并赋予数据库权限。如果 MySQL 服务器就安装在你自己的机器上并且正运转着，你本人就能以 root 身份连接上服务器并为自己新创建一个账户。在下面的示例里，我们给示例数据库 sampdb 增加了一个新

的管理员账户，新账户的用户名是 sampadm、口令是 secret。（你可以把它们改成别的，但要改就得把此处和本书其他有关内容里的用户名和口令都改过来。）

```
% mysql -p -u root
Enter password: ******
mysql> CREATE USER 'sampadm'@'localhost' IDENTIFIED BY 'secret';
Query OK, 0 rows affected (0.04 sec)
mysql> GRANT ALL ON sampdb.* TO 'sampadm'@'localhost';
Query OK, 0 rows affected (0.01 sec)
```

mysql 命令的-p 选项将会让 mysql 提示 MySQL 的 root 用户输入口令。你输入的口令将被显示为一串星号字符，即示例中的******。这里假设你已经为 MySQL 的 root 用户设置了口令。如果你还没有给它设置口令，请在提示 Enter Password:出现后直接按下 Enter 键。不过，root 用户没有口令是很大的安防漏洞，应该尽快给它设置一个。第 12 章讲述了 CREATE USER 和 GRANT 语句的其他信息，MySQL 用户账户的设置和口令的修改。

如果你打算今后就在运行着 MySQL 服务器的这台机器上连接 MySQL，就可以直接套用示例中的语句。这样你就能以用户名 sampadm 和口令 secret 连接上服务器，还能拥有 sampdb 数据库上的全部访问权限。注意：GRANT 语句并不能创建出数据库来（你可以在数据库创建之前赋予权限），我们稍后再来讨论数据库的创建问题。

如果你打算今后使用另一台计算机通过网络来连接 MySQL 服务器，就需要把示例中的 localhost 改为那台计算机的名字。举个例子，如果你将从主机 asp.snake.net 来连接 MySQL 服务器，就得把语句改写为下面这样：

```
mysql> CREATE USER 'sampadm'@'asp.snake.net' IDENTIFIED BY 'secret';
mysql> GRANT ALL ON sampdb.* TO 'sampadm'@'asp.snake.net';
```

如果你无法亲自操作服务器，也不能创建用户，那就得求助于 MySQL 管理员，让他为你建立一个新账户了。如果是这样，你就得把在本书各示例中出现的 sampadm、secret、sampdb 分别替换为管理员分配给你的用户名、口令和示例数据库名。

1.4.3 如何建立和断开与服务器的连接

通过 Unix 系统的 shell 提示符或者通过 Windows 下的 DOS 控制台用命令提示符调用 mysql 程序就能连接上 MySQL 服务器。这个命令如下所示：

```
% mysql options
```

本书使用%作为命令提示符。事实上，%是 Unix 系统的一个标准提示符，另一个是$。在 Windows 下，有 C:>这样的提示符。（输入示例中的命令时，不用再输入提示符。）

这个 mysql 命令行里的 options 部分表示允许是空白。但下面这种形式的命令可能更多见一些：

```
% mysql -h host_name -p -u user_name
```

在执行 mysql 程序时，用不着把全部选项都写出来，但通常至少需要用户给出自己的用户名和口令来。下面是这几个选项的含义和用法。

- -h *host_name*（替换形式：-- host = *host_name*）

  待连接的服务器主机名。如果主机就是运行 mysql 客户程序的那台机器，此选项就可以省略。
- -u *user_name*（替换形式：-- user = *user_name*）

你的 MySQL 用户名。在 Unix 系统上，如果你的 MySQL 用户名与你的登录名完全一样，就可以省略这个选项——mysql 将自动把你的登录名用做你的 MySQL 用户名。

在 Windows 系统上，默认用户名是 ODBC。但这个默认名也许并不归你拥有。你可以通过命令行上的-u 选项明确地给出用户名，也可以通过环境变量 USER 来隐含地给出用户名。比如说，你可以用下面这条 set 命令设定一个名为 sampadm 的用户：

```
C:\> set USER=sampadm
```

如果已经通过 Control Panel（控制面板）中的 System（系统）页面设置了 USER 环境变量，该设置将影响到每一个控制台窗口，你就用不着再从命令提示符发出这个命令了。

- -p（替换形式：--password）

  这个选项的作用是通过 Enter password:让 mysql 提示你输入 MySQL 口令。比如说：

```
% mysql -h host_name -p -u user_name
Enter password:
```

  当看到提示 Enter password:时，请输入你的口令。（你输入的口令将不会显示在屏幕上，以免被你身后的人偷看到。）注意：MySQL 口令并不一定要与 Unix 或 Windows 口令相同。

  如果你省略了-p 选项，mysql 就将认为你不需要口令，也就不会提示你输入它了。

  这个选项的另一种形式是在命令行上直接给出口令，以-p*your_pass*（替换形式：--password=*your_pass*，其中的 *your_pass* 就是你的口令）的形式直接敲入口令。但出于安全方面的考虑，最好别这样做，因为你身后的人会看到它。

  如果你确实想在命令行上直接敲入口令，有一点请特别注意：在 -p 和口令之间不允许有空格存在。-p 选项的这一特点（不需要输入空格）很容易与输入-h 和-u 选项时的情况弄混，无论选项与口令之间是否有空格，-h 和-u 都与其后的口令关联。

假设 MySQL 用户名和口令分别是 sampadm 和 secret，那么，如果 MySQL 服务器就运行在同一台主机上，就可以省略-h 选项和 mysql 命令而像下面这样去连接服务器：

```
% mysql -p -u sampadm
Enter password: ******
```

输入完这条命令后，mysql 将显示 Enter password:以提示你应输入口令。此时敲入口令（输入的 secret 将在屏幕上显示为 6 个星号字符******）。

如果一切正常，mysql 将显示欢迎消息和一个 mysql>提示以表明它在等你发出 SQL 查询命令。下面是完整的操作过程：

```
% mysql -p -u sampadm
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 13762
Server version: 5.0.60-log

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

如果 MySQL 服务器运行在另一台机器上，就必须通过-h 选项来指定那台机器的主机名。假设那台主机的名字是 cobra.snake.net，就必须使用下面这样的命令：

```
% mysql -h cobra.snake.net -p -u sampadm
```

为简洁起见，我将在后面内容里出现的 mysql 命令行上省略-h、-p 和-u 选项，但大家在做练习或实际工作中请不要忘了输入它们。在运行其他 MySQL 程序（如 mysqlshow）时，你将用到这些选项。

在连接上 MySQL 服务器之后，我们随时都能通过 quit 命令来结束这次会话。如下所示：

```
mysql> quit
Bye
```

你也可以通过敲入字符串 exit 或\q 来退出。在 Unix 系统下，利用组合键 Ctrl-D 也可以退出。

在刚开始学习 MySQL 时，很多人都觉得它的安防系统给自己添了不少麻烦——你必须有足够的权限才能创建和访问数据库，你必须正确地给出用户名和口令才能连接上服务器。但在抛开教科书里的示例数据库而开始使用自己的数据记录之后，这些人的看法就会迅速改变，转而感激 MySQL 有这样一个能防止别人搜索或者（更糟糕）破坏自己信息的安防系统。

有些方法能让你把工作环境提前设置好，使你不必在每次运行 mysql 的时候都不得不在命令行输入一大堆的连接参数，1.5 节将会讨论这个问题。简化服务器连接过程最常见的办法是把连接参数存放到一个选项文件里。如果你想现在就去建立一个这样的文件，不妨直接跳到 1.5 节。

1.4.4 执行SQL语句

连接上服务器以后，你就可以发出查询命令让服务器执行了。本节将介绍一些与 mysql 交互的一般原则。

利用 mysql 来进行数据库查询很简单：先敲入有关命令，再在命令的末尾敲入一个分号字符（;）表示语句结束，再按下 Enter 键就行了。在你完成查询命令的输入之后，查询命令将由 mysql 送往服务器执行，服务器对查询进行处理并把其结果回送给 mysql，最后由 mysql 把查询结果显示在屏幕上。

下面这个简单的查询命令将让你看到系统的当前日期和时间：

```
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2008-03-21 10:51:23 |
+---------------------+
1 row in set (0.00 sec)
```

除使用分号外，终止语句的另一种方法是使用\g（表示 go）：

```
mysql> SELECT NOW()\g
+---------------------+
| NOW()               |
+---------------------+
| 2008-03-21 10:51:28 |
+---------------------+
1 row in set (0.00 sec)
```

也可以使用\G，竖直排列显示结果，每行一个值：

```
mysql> SELECT NOW(), USER(), VERSION()\G
*************************** 1. row ***************************
    NOW(): 2008-03-21 10:51:34
   USER(): sampadm@localhost
VERSION(): 5.0.60-log
1 row in set (0.03 sec)
```

如果查询命令的输出行比较短，以\G 作为查询命令结束符的效果还不太明显。可万一输出行比较长，在屏幕上显示为好几行的时候，\G 结束符就能使屏幕输出内容更容易阅读一些。

如上所示，mysql 把查询结果、构成本次查询结果的数据行的个数以及用来处理本次查询所花费的时间依次显示出来。为简洁起见，我将在本书后面的示例里省略用来给出查询结果数据行总数的那一行。

因为 mysql 必须等待语句结束符，所以我们用不着把查询命令完整地写在同一个命令行上，我们可以用多个命令行来输入一条查询命令，如下所示：

```
mysql> SELECT NOW(),
    -> USER(),
    -> VERSION()
    -> ;
+---------------------+--------------------+------------+
| NOW()               | USER()             | VERSION()  |
+---------------------+--------------------+------------+
| 2008-03-21 10:51:37 | sampadm@localhost  | 5.0.60-log |
+---------------------+--------------------+------------+
```

请注意，当我们输入查询命令的第一行时，提示符将从 mysql>变为->。这是在提醒你 mysql 认为你仍要继续输入查询命令。这种反馈非常重要，如果你遗漏了应该添加在查询命令末尾的分号，发生了变化的提示符将提醒你注意 mysql 仍在等待你继续输入。不然，就可能你这边在为 MySQL 经过了这么长的时间还没有完成你的查询而疑惑而烦躁，mysql 那边却在耐心地等待你把这条查询命令输入完！（mysql 还有几个别的提示符，我们将在附录 F 里介绍它们。）

如果你已经输入了好几行查询命令却不想执行它，可以敲入\c 来清除（即取消）它，如下所示：

```
mysql> SELECT NOW(),
    -> VERSION(),
    -> \c
mysql>
```

请注意，提示符将变回为 mysql>以表明 mysql 程序准备接收下一条查询命令。

与将一条语句输入成多行相反的是，在一行上输入多条语句，用终止符分隔。

```
mysql> SELECT NOW();SELECT USER();SELECT VERSION();
+---------------------+
| NOW()               |
+---------------------+
| 2008-03-21 10:52:31 |
+---------------------+
+-------------------+
| USER()            |
+-------------------+
| sampadm@localhost |
+-------------------+
+------------+
| VERSION()  |
+------------+
| 5.0.60-log |
+------------+
```

在大多数情况下，查询命令允许以大写字母、小写字母或者大小写字母混用的形式来输入。比如

说，下面这几条查询命令就是等效的（虽然大小写不同）：

```
SELECT USER();
select user();
SeLeCt UsEr();
```

在后面的示例语句里，我们将用大写字母来写出 SQL 关键字和函数名，用小写字母来写出数据库、数据表和数据列的名字。

如果你想在语句里使用函数，请千万记住这样一件事：在函数名与它后面的括号中间不允许出现空格。有时空格会导致语法错误。

你可以把查询命令提前保存在一个文件里，再创建一个 SQL 脚本让 mysql 从那个文件而不是键盘来读取语句。这要用到 shell（操作系统的命令解释器）的输入重定向功能。比如说，如果把语句提前保存在一个名为 myfile.sql 的文件里，就可以像下面这样来执行它们（要指定任何必需的连接参数选项）：

```
% mysql < myscript.sql
```

这个文件的名字可以随便起。我喜欢给它们加上一个“.sql”后缀以表明这个文件里存放的是 SQL 语句。

这种利用 shell 的重定向功能来调用 mysql 的做法将在 1.4.7 节出现，我们将用这种办法把数据录入到 sampdb 数据库里。与一条一条地手工敲入一大堆 INSERT 语句相比，让 mysql 从某个文件里来读取它们要方便和快捷得多。

1.4 节的后续内容里有很多为大家练习而准备的 SQL 语句，它们以 mysql>提示符为标志，语句在提示符后面，且基本上都包括查询结果。如果你输入的语句与示例中的一模一样，那它们的查询结果也应该完全相同。但我在某些查询命令的前面没有给出提示符，它们主要用来阐明某个概念，不要求读者真的去执行它们。（当然，如果你愿意，试试它们也没什么坏处。但如果你打算用 mysql 来这样做，请不要忘记在它们的末尾加上一个分号作为结束符。）

1.4.5 创建数据库

我们的学习将从创建 sampdb 示例数据库与其中的数据表、把有关数据录入各数据表、利用这些数据表里的数据完成一些简单的查询任务开始逐步展开。使用数据库有以下几个步骤。

(1) 创建（初始化）一个数据库。

(2) 在数据库里创建各种数据表。

(3) 对数据表里的数据进行插入、检索、修改、删除等操作。

数据库上最常见的操作是对现有数据进行检索，比较常见的操作是插入新数据、修改或者删除现有数据，比较不常见的操作是创建数据表，最不常见的操作是创建数据库。但因为我们的学习要从一穷二白的状况开始，所以我们反而要从最不常见的数据库创建操作入手，再经过创建数据表和录入原始数据等步骤之后才能完成最常见的数据库操作——数据的检索。

创建新数据库的做法是：先用 mysql 连接上服务器，再用一条 CREATE DATABASE 语句给出新数据库的名字：

```
mysql> CREATE DATABASE sampdb;
```

只有在创建了 sampdb 数据库之后，才能创建该数据库里的各个数据表并对这些数据表的内容进

行各种操作。

那么，创建一个数据库是否就意味着把它选定为当前的默认数据库呢？答案是“否”。你可以用下面这条语句去核实一下：

```
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| NULL       |
+------------+
```

Null 意味着没有选择数据库。如果想把 sampdb 设置为当前的默认数据库，就需要发出一条 USE 语句：

```
mysql> USE sampdb;
mysql> SELECT DATABASE();
+------------+
| DATABASE() |
+------------+
| sampdb     |
+------------+
```

选定默认数据库的另一种办法是在启动 mysql 的命令行上给出该数据库的名字：

```
% mysql sampdb
```

事实上，今后大家在选定数据库时用得最多的可能还是这后一种办法。如果需要指定服务器连接参数，请在命令行指定。比如说，下面两条命令将把 sampadm 用户连接到本地主机（如果没有指定主机名字，就是默认的）的 sampdb 数据库上去：

```
% mysql -p -u sampadm sampdb
```

如果需要连接到在远程主机上运行的 MySQL 服务器，就应在命令行指定主机：

```
% mysql -h cobra.snake.net -p -u sampadm sampdb
```

如果没有特别说明，以后的示例都将假定你在启动 mysql 时已经在命令行上把 sampdb 数据库选定为当前的默认数据库。如果你在启动 mysql 时忘了在命令行选定这个数据库，请在 mysql>提示符处发出一条 USE sampdb 语句。

1.4.6　创建数据表

在本节里，我们将创建示例数据库 sampdb 里的各个数据表。首先创建“美国历史研究会”场景所需要的各种数据表，然后再创建“考试记分项目”所需要的各种数据表。在这个部分，有些数据库图书会大谈特谈数据库的分析与设计、实体联系图、规范化过程等概念。有些书专门讲解这些概念，而本书只讨论我们的数据库应该是什么样子——它应该包含哪些数据表，各数据表应该有什么样的内容，以及数据应该如何表示。

这里所选定的数据表示方式并不是绝对的，换在另一种场合，你可能会选用另一种方式来表示类似的数据——这应该由应用项目的具体要求和数据的具体用途来决定。

1. 美国历史研究会的数据表

美国历史研究会场景需要的数据表相当简单，如下所示。

- ❑ president（总统）数据表。用来保存关于美国历届总统的描述性记录。我们将利用它在研究会的网站上提供历史知识在线小测验（会刊儿童栏目中的打印的小测验的交互式模拟版）。
- ❑ member（会员）数据表。用来保存每一位会员的个人最新资料。研究会将利用这个数据表来完成制作会员名录（纸印刷品以及网上版本）、自动提醒会员续交会费等工作。

● president数据表

president 数据表比较简单，所以我们先来讨论它。这个表里包含着关于美国历届总统生平的基本信息。

- ❑ **姓名**。在数据表里，有好几种办法可以用来表示人的姓名。比如说，我们既可以把姓名保存在同一个数据列里，也可以把人的姓氏和名字分别保存在不同的数据列里。用同一个数据列来保存姓名当然要简单一些，但这种做法有一定的局限性，如下所示。
  - ■ 如果先输入名字，就无法按姓氏进行排序。
  - ■ 如果先输入姓氏，就无法按名在前姓在后的（英语国家）习惯顺序来显示它们。
  - ■ 很难对姓名进行查找。比如说，如果你想查找某个姓氏，就必须使用一个匹配模板（pattern）来查找与之匹配的姓名。与直接查找姓氏的做法相比，这种做法效率低且速度慢。

  为了避开这些限制，president 数据表将把总统们的姓氏和名字分别保存在不同的数据列里。

  我们把总统们的中间名或第一个名字也安排在名字数据列里。因为我们不太可能对中间名（也不太可能对第一个名字）排序，所以这应该不会影响到我们将对总统姓名进行的排序。这也不会影响到姓名的显示，因为无论是按“Bush, George W.”还是按“George W. Bush”的格式来显示，姓名里的中间名总是紧跟在名字的后面。

  还有一个问题需要考虑。有位总统（如 Jimmy Carter）的姓名后面还有一个“Jr.”。应该把这个东西放到哪儿呢？根据英语习惯，这位总统的名字既可以写成“James E. Carter, Jr.”，也可以写成“Carter, James E., Jr.”。这个“Jr.”只能出现在整个姓名的末尾，无法与名字或姓氏结合在一起。因此，我们决定另建一个数据列来保存这种姓名后缀。这种情况请大家务必注意：即使只有一个特例值，也会影响到你在选择数据表示形式时的决策。它同时还证明了这样一条经验：应该在事先对将被保存到数据库里去的数据值做尽可能深入的了解。如果你没有在事先对这些问题做周密的考虑，就可能会在启用数据库之后还不得不再去修改数据库结构。这种事情虽说不是什么灾难，但还是从一开始就尽量避免为好。
- ❑ **出生地（州和城市）**。与姓名的情况类似，这些信息也是既可以保存在同一个数据列，也可以保存在多个数据列。保存在同一个数据列的做法要简单些，但把它们分别保存在不同的数据列将使你的某些工作更容易完成。比如说，如果把州名与城市名分开放置，诸如“出生在某个州的总统有多少”之类的问题就更容易查询出来。
- ❑ **出生日期和逝世日期**。这里需要考虑的特殊情况是：不能要求必须填上逝世日期，因为有些总统还依然健在呢。MySQL有一个专用的特殊值NULL来对付这种“无数据”的情况，所以我们将在逝世日期列里用这个值来表示“依然健在”的情况。

● member数据表

从每条记录都保存着某个人的个人资料的角度看，用来存放美国历史研究会会员个人资料的 member 数据表与刚才介绍的 president 数据表差不多。但每个 member 数据表里还包含其他一些数据列，如下所示。

- **姓名**。我们将沿用president数据表的3个数据列（姓氏、名字和姓名后缀）表示法。
- **ID编号**。每个会员都会在其会员资格初次生效时分配到一个独一无二的编号。研究会以前从没对会员进行过编号，但既然打算从现在起对会员进行更系统化的管理，所以眼下正是一个好时机。（我希望大家会不断发现MySQL的好处并琢磨出会员资料更多的用途。当你需要把member数据表和其他与会员有关的数据表联系起来时，会员编号用起来比姓名要简便得多）。
- **失效日期**。会员必须定期续费才能保证其会员资格不会过期失效。在别的项目里，你可能需要把这个日期表示为“上次交费日期”，但这对美国历史研究会的情况不适用。会员资格的有效期是一个可变的数字（可以是一年、两年、三年或者五年），而“上次交费日期”并不能告诉你某个会员必须在何时交纳下一期会费。所以我们将保存会员资格的失效日期。此外，研究会还有一个终身会员制度。虽然可以用一个遥远的未来日期来代表这种情况，但特殊的NULL值更理想，因为用“无数据”来代表“永不失效”是非常合乎逻辑的。
- **电子邮件地址**。电子邮件地址将使兴趣相同的会员更容易交流。对于身为研究会秘书的你来说，这些地址将使你能够以电子方式向会员发出续费通知而不必再依靠普通信件。与跑到邮局去寄信相比，这种做法既方便又省钱。你还可以利用电子邮件把会员们的当前个人资料发送给他们做必要的修改。
- **邮政地址**。这是为那些无法通过电子邮件进行联络（或者没有回复你电子邮件）的会员而准备的。我们将把街道地址、城市名、州名和邮政编码分别保存在不同的数据列里。

  我们这里要假设全体会员都居住在美国。当然了，对于那些在世界各地都有会员的组织机构来说，这个假设过于简单了。如果涉及多个国家的地址，你就必须研究各种地址格式。比如说，邮政编码并没有一项国际标准，还有些国家被划分为省而不是州。
- **电话号码**。这些信息与地址列的作用相似，都是为了与会员联络。
- **会员兴趣关键字**。研究会的每位会员都对美国历史感兴趣，但他们的兴趣却可能集中在某些特定的历史时期上。这个数据列就是用来记录这种特殊兴趣的。会员可以利用这些信息来寻找与自己兴趣相同的其他会员。（严格说来，建立一个单独的表会更好，其中的行由一个关键字和相关成员ID组成。即便这也有弊端，我不想在这儿谈。）

● 美国历史研究会各数据表的创建

下面，我们将开始创建“美国历史研究会”的各种数据表。我们要用 CREATE TABLE 语句来完成这一工作，这条语句的格式是：

```
CREATE TABLE tbl_name (column_specs);
```

其中，*tbl_name* 是给数据表起的名字，*column_specs* 则是该数据表里的各个数据列以及各种索引（如果有的话）的定义。索引能够加快信息的检索速度；我们将在第 5 章中介绍它们。

下面是对应于 president 数据表的 CREATE TABLE 语句：

```
CREATE TABLE president
(
  last_name  VARCHAR(15) NOT NULL,
  first_name VARCHAR(15) NOT NULL,
  suffix     VARCHAR(5) NULL,
  city       VARCHAR(20) NOT NULL,
  state      VARCHAR(2) NOT NULL,
  birth      DATE NOT NULL,
```

```
  death       DATE NULL
);
```

执行这条语句的方法有好几种。可以手动输入，也可以使用sampdb发行版的create_president.sql文件中预先写好的语句。

如果你打算亲自输入这条语句，请先用下面的命令来启动mysql客户程序，并把数据库sampdb设置为当前的默认数据库：

```
% mysql sampdb
```

然后再敲入上面的CREATE TABLE语句。不要漏掉语句末尾的分号，这样才能让mysql程序知道这条语句的结束位置。有缩进没有关系，你不需要在同一处换行。例如，你可以在同一行输入一条语句。

如果打算利用一个预先写好的描述文件来创建president数据表，可以使用sampdb发行版本里的create_president.sql文件。你可以在系统安装这个发行版本时所创建的sampdb子目录里找到这个文件。先切换到那个子目录，然后再执行下面这条命令：

```
% mysql sampdb < create_president.sql
```

不管如何启动mysql客户程序，都不要忘记在命令行里的命令名称之后加上必要的连接参数（主机名、用户名、口令等）。

CREATE TABLE语句中的数据列定义由以下几部分组成：数据列的名字、数据类型（这个数据列是用来保存哪种数据的）和一些属性。

president数据表用到了两种数据类型：VARCHAR(*n*)和DATE。VARCHAR(*n*)的意思是这个数据列里存放着长度可变的字符（串）值，最多有*n*个字符。这个*n*要由你根据对字符串数据长度的预估来选定。比如说，我们把state数据列定义为VARCHAR(2)类型，这是根据美国州名都可以被缩写为两个字母的事实而确定的。其他字符串类型的数据列所要容纳的值可能会比较长，所以它们要宽一些。

我们用到的另一种列类型是DATE。这种类型的数据列用来保存日期值，这用不着多说，但大家千万要注意日期值的表示格式。MySQL要求日期被表示为'*CCYY-MM-DD*'的格式，其中*CC*、*YY*、*MM*、*DD*分别代表世纪、年份、月份和日期。这是SQL中规定的日期表示标准（也叫做ISO 8601格式）。比如说，2002年7月18日在MySQL里必须被表示为'2002-07-18'而不是'07-18-2002'或'18-07-2002'。

president数据表还用到了两种数据列属性：NULL（表示无数据）和NOT NULL（表示不得为空）。表中的大部分数据列都具有NOT NULL属性，因为我们必须在其中填上数据。具有NULL属性的数据列有两个，一个是suffix（姓名后缀，大多数总统的姓名里都没有后缀），另一个是death（逝世日期，有些总统还活着，用不着填逝世日期）。

下面是我们用来创建member数据表的CREATE TABLE语句：

```
CREATE TABLE member
(
  member_id  INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (member_id),
  last_name  VARCHAR(20) NOT NULL,
  first_name VARCHAR(20) NOT NULL,
  suffix     VARCHAR(5) NULL,
  expiration DATE NULL,
  email      VARCHAR(100) NULL,
  street     VARCHAR(50) NULL,
```

```
  city        VARCHAR(50) NULL,
  state       VARCHAR(2) NULL,
  zip         VARCHAR(10) NULL,
  phone       VARCHAR(20) NULL,
  interests   VARCHAR(255) NULL
);
```

你可以手动输入这些语句，也可以利用 sampdb 发行版本里的预编写文件 creat_member.sql，其中包含用于 member 表的 CREATE TABLE 语句。执行下面的命令：

```
% mysql sampdb < create_member.sql
```

在 member 数据表里，大部分数据列的类型都是可变长度的字符串。只有两个数据列例外：用来保存会员编号的 member_id 和用来保存失效日期的 expiration。

为了避免混淆不同的会员，数据列 member_id 里的值必须是独一无二的。这正是 AUTO_INCREMENT 数据列大显身手的地方——当我们往 member 数据表添加新记录时，MySQL 能在 member_id 列自动生成一个唯一的会员编号。虽然我们将要放到 member_id 数据列里的只是些数字，但它的定义却包含了好几个部分，如下所示。

- INT。表示这个数据列将用来保存整数值（没有小数部分的数字）。
- UNSIGNED。不允许出现负数。
- NOT NULL。必须填有数据，不得为空。（这意味着每个会员必须有一个会员号。）
- AUTO_INCREMENT。这是MySQL里的一个特殊属性。它表示数据列里存放的是序列编号。AUTO_INCREMENT机制的工作原理是这样的：当我们往member数据表里插入数据记录时，如果没有给出member_id列的值（或者给出的值是NULL），MySQL将自动生成下一个编号并赋值给这个数据列。这样，为新会员分配会员号的工作就简单了，因为MySQL可以替我们完成。

PRIMARY KEY 子句表示需要对 member_id 数据列创建索引以加快查找速度，同时也要求该数据列里的各个值都必须是唯一的。后者正好满足了我们对会员 ID 的编号要求，因为我们绝不希望误把同一个会员号分配给两个会员。此外，MySQL 本身也要求每一个具备 AUTO_INCREMENT 属性的数据列必须拥有某种形式的唯一化索引，若没有，数据表的定义就不合法。任何 PRIMARY KEY 列都必须是 NOT NULL，所以，如果在 member_id 定义中忽略了 NOT NULL，MySQL 将自动添加上去。

如果你现在还不能理解 AUTO_INCREMENT 和 PRIMARY KEY 的含义与作用，不妨把它们想象成一种用来为生成带索引的会员号的魔术好了。我们真正关心的是那些会员号是否都是唯一的，它们到底等于多少并不重要。（有关 AUTO_INCREMENT 数据列的使用请参见第 3 章。）

expiration 列是一个 DATE。它可以为 NULL 值，所以其默认值为 NULL。NULL 意味着可以不输入数据。原因就是前面提到的，expiration 可以为 NULL，表明成员具有终身会员资格。

现在，既然已经让 MySQL 创建了几个数据表，我们就该去检查一下它做得怎么样。在 mysql 里，我们可以用下面这条命令来查看 president 表的结构：

```
mysql> DESCRIBE president;
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| last_name  | varchar(15) | NO   |     |         |       |
| first_name | varchar(15) | NO   |     |         |       |
| suffix     | varchar(5)  | YES  |     | NULL    |       |
| city       | varchar(20) | NO   |     |         |       |
```

```
| state      | varchar(2)  | NO   |     |         |       |
| birth      | date        | NO   |     |         |       |
| death      | date        | YES  |     | NULL    |       |
+------------+-------------+------+-----+---------+-------+
```

如果你发出的是 DESCRIBE member 命令，mysql 就将显示有关 member 数据表的类似信息。

DESCRIBE 是一个非常有用的命令，尤其是当你想不起数据列的名字、类型或数据长度等细节的时候。你还可以利用这条命令来查看各数据列在数据行里的存储先后顺序，这个顺序很重要，INSERT 或 LOAD DATA 等语句要求各数据列的值必须按它们默认的存储顺序依次列出。

能够用 DESCRIBE 命令查出来的信息也可以通过别的手段获得。你可以把它简写为 DESC，也可以把它写成 EXPLAIN 或 SHOW 语句。下面这些语句的作用是相同的：

```
DESCRIBE president;
DESC president;
EXPLAIN president;
SHOW COLUMNS FROM president;
SHOW FIELDS FROM president;
```

这些语句还允许你把输出内容限制为指定的数据列。比如说，如果你在 SHOW 语句的末尾加上一个 LIKE 子句，就只能看到与给定模板相匹配的那几个数据列的有关信息，如下所示：

```
mysql> SHOW COLUMNS FROM president LIKE '%name';
+------------+-------------+------+-----+---------+-------+
| Field      | Type        | Null | Key | Default | Extra |
+------------+-------------+------+-----+---------+-------+
| last_name  | varchar(15) | NO   |     |         |       |
| first_name | varchar(15) | NO   |     |         |       |
+------------+-------------+------+-----+---------+-------+
```

DESCRIBE president '%name'是等效的。这里使用的百分号（%）是一个特殊的通配符，1.4.9 节的第 7 小节将介绍它。

SHOW 语句还有其他几种用法，可以用来从 MySQL 获取各种信息。SHOW TABLES 能够列出当前默认数据库里的数据表。例如，我们已经在 sampdb 数据库里创建了两个数据表，于是输出为：

```
mysql> SHOW TABLES;
+-------------------+
| Tables_in_sampdb  |
+-------------------+
| member            |
| president         |
+-------------------+
```

另外，SHOW DATABASES 能够列出当前连接的服务器上的数据库，如下所示：

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| menagerie          |
| mysql              |
| sampdb             |
| test               |
+--------------------+
```

服务器不同,这条语句列出来的数据库清单也就不同,但你至少应该能看到 information_schema 和 sampdb.information_schema。sampdb 数据库是我们刚创建的。你还会看到 test 数据库，它是在 MySQL 安装过程中创建的。根据你的访问权限，你还会看到 mysql 数据库，其中存放着各种用来控制 MySQL 访问权限的权限分配表。

客户程序 mysqlshow 提供命令行接口，用 SHOW 语句能查看到的信息也都能用 mysqlshow 程序查看到。

不带参数的 mysqlshow 程序将列出一份数据库清单：

```
% mysqlshow
+--------------------+
|     Databases      |
+--------------------+
| information_schema |
| menagerie          |
| mysql              |
| sampdb             |
| test               |
+--------------------+
```

如果给它加上一个数据库名，mysqlshow 将列出一份给定数据库里的数据表清单：

```
% mysqlshow sampdb
Database: sampdb
+-----------+
|  Tables   |
+-----------+
| member    |
| president |
+-----------+
```

如果同时给出一个数据库名和数据表名，mysqlshow 将显示那个数据表里各数据列的信息——就像 SHOW FULL COLUMNS 语句那样。

2. 考试记分项目的数据表

要想确定考试记分项目需要用到哪些数据表，先要弄清楚怎样用纸质记分簿来记录考生成绩。请看图 1-2，假设这是纸质记分簿里的某一页，上面是一个记有考试分数的表格，其中还包含其他一些使考试分数更有意义的信息。学生的姓名和 ID 号列在表格的左侧（为简洁起见，我只列出了 4 位学生），考试或测验的举行日期则列在表格的顶部。表格显示，9 月的 3、6、16、23 日进行了测验，9 月 9 日和 10 月 1 日进行了考试。

学生		分数						
		Q	Q	T	Q	Q	T	
ID	姓名	9/3	9/6	9/9	9/16	9/23	10/1	...
1	Billy	14	10	73	14	15	67	...
2	Missy	17	10	68	17	14	73	...
3	Johnny	15	10	78	12	17	82	...
4	Jenny	14	13	85	13	19	79	...
...	...	...	...	...	...	...	...	...

图 1-2　纸质记分簿里的某一页

要想把这些信息记录到一个数据库里，就需要一个 score 数据表。那么，这个数据表里的各条记

录应该包含哪些信息呢？这个问题不难回答。在每个数据行里，需要列出学生的姓名、考试或测验的日期和学生的考试分数。图 1-3 给出了数据表里的一些考试分数。（注意，日期是按 MySQL 的日期表示法'CCYY-MM-DD'格式写出来的。）

score 数据表

name	date	score
Billy	2008-09-23	15
Missy	2008-09-23	14
Johnny	2008-09-23	17
Jenny	2008-09-23	19
Billy	2008-10-01	67
Missy	2008-10-01	73
Johnny	2008-10-01	82
Jenny	2008-10-01	79

图 1-3 最初的 score 数据表

可是，如此得到的数据表是有问题的，它丢失了某些信息。比如说，仔细看看图 1-3 中的行就会发现，我们无法区别考试分数与测验分数。一般说来，在评定学生们的期末总成绩时，考试分数与测验分数的比重是有一定区别的，所以有必要知道考分类型。当然了，我们可以根据某给定日期的分数范围（测验分数通常要比考试分数在数值上低很多）来推测出这个类型，但这种不用数据明确表明而纯粹依靠推理的做法会带来问题。

要想在每行记录里把考分类型区别开还是有办法的，例如，可以像图 1-4 那样给 score 数据表增加一个数据列，并用 T 或 Q 来分别代表 test（考试）或 quiz（测验）。这种做法的好处是考分类型能直接体现在数据上，坏处是这部分信息有些冗余。看看那些日期相同的行就能发现，考分分类(category)栏里的值全都一模一样。在 9 月 23 日，所有考分的类型全都是 Q；到了 10 月 1 日，所有考分的类型又全都是 T。这可有点太啰嗦了。要是按这种办法来记录学生们的考试分数，我们不光要反复多次地输入一个相同的日期，还不得不反复多次地输入一个相同的考分类型。有谁愿意反复输入这么多的冗余信息呢？

score 数据表

name	date	score	category
Billy	2008-09-23	15	Q
Missy	2008-09-23	14	Q
Johnny	2008-09-23	17	Q
Jenny	2008-09-23	19	Q
Billy	2008-10-01	67	T
Missy	2008-10-01	73	T
Johnny	2008-10-01	82	T
Jenny	2008-10-01	79	T

图 1-4 修改后的 score 数据表，增加了一个 type 列

我们应该想出一个更好的办法。与其把考分类型放到 score 数据表里，不如把它与考试日期对应起来。可以把考试日期列成一个表，再把各日期里发生的“考试事件”（测验或考试）记录在这个表里。这样，只要根据 score 表里的日期在 grade_event 表里查出当天的考试事件类型，我们就能知道某个分数是来自测验还是来自考试。图 1-5 给出了这种思路下的数据表布局，并以 2008 年 9 月 23 日为例画出了 score 表与 grade_event 表的对应关系。根据 score 数据表里的日期，我们从 grade_event 数据表里查出当天举行的是一次测验，所以 score 表里的那个分数是一次测验成绩。

score 数据表

name	date	score
Billy	2008-09-23	15
Missy	2008-09-23	14
Johnny	2008-09-23	17
Jenny	2008-09-23	19
Billy	2008-10-01	67
Missy	2008-10-01	73
Johnny	2008-10-01	82
Jenny	2008-10-01	79

grade_event 数据表

date	category
2008-09-03	Q
2008-09-06	Q
2008-09-09	T
2008-09-16	Q
2008-09-23	Q
2008-10-01	T

图 1-5　通过 date 列相联系的 score 与 grade_event 数据表

与通过推测来判断考分类型的做法相比，新办法要好得多了，因为现在能够从记录在数据库里的数据直接得出考试分数的类型。与把考分类型直接记录在 score 数据表里的做法相比，新办法也要好得多——我们总共只需记录一次考分类型，不必再为每个考分都要记录一次了。

不过，现在需要把多个数据表的信息结合起来才行。也许你和我一样，在第一次听说这种事的时候，可能会想："嘿，这个主意可真够酷的。可这么多的数据表，想查什么东西会不会太费事？这会不会把事情搞得更复杂呢？"

从某种意义上讲，这种担心是有道理的。记录两个表当然要比记录一个表复杂。可仔细看看当初的记分簿（如图 1-2 所示），你不是已经在记录两套信息了吗？请注意以下两个事实。

- 把考试分数记录在表格的每一小格里，这些小格按学生姓名和考试日期排列（按姓名，由上往下排列；按日期，由左往右排列）。这正是我刚才所说的两套信息中的一套，与这套信息相对应的是 score 数据表里的内容。
- 你是怎么知道各日期所代表的事件类型的呢？你在记分簿里是这样做的：在日期的上面写上一个 T 或 Q，在表格的顶部把考试日期与考试类型关联起来。这正是我刚才所说的两套信息中的第二套，与这套信息相对应的是 grade_event 数据表里的内容。

换句话说，也许你本人还没有意识到这一点，但你在记分簿里做的事与我把信息放到两个数据表里的情况并没有什么差异。即便是有差异，也只是纸质记分簿里的两套信息没有明确地分别放置而已。

记分簿表格的例子体现出了人们对信息的思维方式，也反映出这样一个问题：把信息妥善地放到数据库里去并不是一件简单的事情。在日常生活中，人们习惯于把不同信息综合起来并作为一个整体来考虑。但数据库毕竟不是人类的大脑，这正是它们看起来过于人工化和不太自然的原因之一。习惯于把信息综合在一起的思维特点使我们有时很难意识到自己正使用着多种的信息而非一种。因此，以数据库系统的方式来表达数据往往很有挑战性。

图 1-5 里的 grade_event 数据表还隐含了这样一个要求：date 列里的日期必须是独一无二的。因为每一个日期都将被用来联系 score 和 grade_event 数据表里的某些数据记录。换句话说，它要求你不得在同一天进行两场测验（或者一次测验加一次考试）。如果你这样做了，那么，score 数据表里将会有两组考分记录、grade_event 数据表里将会有两条类型记录，都对应于同一个日期。这意味着通过日期的匹配关系来联系 score 记录和 grade_event 记录的做法将难以为继。

假如你每天最多只进行一场考试，那么这个问题就不成其为问题。但一天两场考试的情况真的永远都不会发生的吗？也许如此，心地善良的你应该不会对学生们苛刻到要对他们进行一天两场考试的程度。不过（希望大家别怪我多嘴），虽然经常有人说"这种奇怪的事情永远也不会发生"，可奇怪的事情却真的在某个时刻发生了。为了弥补这一漏洞，这些人就不得不加班加点地去重新设计数据表。

与其临时抱佛脚，不如防患于未然。提前预见到可能出现的各种问题并准备好应对措施将为你减少很多麻烦。因此，还是现在就对“你会记录同一天里的两组考试分数”的情况作一下分析比较好。应该如何解决这个问题呢？别担心，随着讨论的深入，这个问题将迎刃而解。只要对有关数据的布局结构作一个小小的改动，在同一天发生多次考试事件的事情就不再会引起麻烦。这些改动如下所示。

(1) 在 grade_event 表里增加一个数据列，利用它给 grade_event 表里的各个记录分配一个唯一的编号。从效果上讲，这等于是给各次考试事件分别赋予了一个唯一的 ID 编号。我们就给新增的这个数据列起名为 event_id（意思是事件编号）好了。（虽然看着有点奇怪，可这一做法却并不是什么新点子，图 1-2 中的记分簿表格其实已经用到了这个东西。记分簿表格分数记录部分的列序号就相当于这里的事件编号。虽说你没有把各列的序号明确地写出来并标明是“event ID”，但它的的确确存在。）

(2) 在把考试分数记到 score 数据表里去的时候，用考试事件的 ID 来代替考试日期。

完成上述改动后，我们得到了图 1-6 所示的结果。现在，score 和 grade_event 表必须用 event_id 而不是 date 来联系。你用 grade_event 表不仅能查出考分的类型，还能查出它具体发生在哪一天。最重要的是，grade_event 表中必须具备唯一性的不再是日期，而是事件编号。这意味着即使在一天之内进行了十几场考试和测验，也能条理清晰地把各场的分数全都记录下来。（你的学生肯定害怕听到这个消息。）

score数据表

name	event_id	score
Billy	5	15
Missy	5	14
Johnny	5	17
Jenny	5	19
Billy	6	67
Missy	6	73
Johnny	6	82
Jenny	6	79

grade_event数据表

event_id	date	category
1	2008-09-03	Q
2	2008-09-06	Q
3	2008-09-09	T
4	2008-09-16	Q
5	2008-09-23	Q
6	2008-10-01	T

图 1-6　通过事件编号列相联系的 score 与 grade_event 数据表

应该承认，图 1-6 里的表格不如前面那几个看起来顺眼。score 表变得越来越抽象，数据列的含义也越来越不容易看懂。请看图 1-4 里的 score 表，那里面既有考试日期又有考分类型，让人一眼就能看明白。但在图 1-6 里，这两个数据列却不见了，我们看到的是一个高度抽象化的信息表示形式。谁会愿意看一个包含 event_id 的 score 表呢？它对我们而言没多大意义。

此时此刻，我们来到了一个十字路口。此前，大家对电子化的考试记分系统充满希望，觉得很快就能从繁琐的评分工作中解脱出来。但在看过上面的讨论后，却发现单是把信息放到数据库里去的事就已经很不容易做到最好了。高度抽象的信息与它们所代表的事物似乎毫无联系，这往往会让人们产生一种畏难情绪。

这很自然地引出了一个问题：“干脆不用数据库会不会更好？也许 MySQL 不适合我。”我的回答大家肯定都能猜到，要不这本书就不会有这么厚了。但对读者来说，在项目开工前多考虑几种办法总是好的。你们应该问自己：是使用 MySQL 这样的数据库系统好，还是使用电子表格（spreadsheet）等其他办法好？从一方面看，

- 记分簿由行和列构成，电子表格也是如此，它们二者在概念和外观上都很相似；
- 电子表格程序能够进行计算，所以使用计算字段进行分数统计工作不难完成。测验分数和考试分数可能不太容易按不同权重来统计，但肯定有办法解决。

从另一方面看，如果你想只对一部分数据进行操作（比如只统计测验分数，或者只统计考试分数），或者想进行某种对比分析（比如男生与女生的成绩对比），或者想灵活地汇总和显示各种统计信息，事情就不同了。这些工作电子表格都不擅长，关系数据库系统则能大显身手。

往开处想，关系数据库中数据的高度抽象化也不是什么不得了的事。你只是在数据库建立之初需要考虑信息在数据库里的表示方式，按照最符合你目标的方式来设置它们。在数据库建立起来以后，信息数据的提取和显示工作将由数据库引擎按一定的逻辑来完成，你看到的是有意义的资料，而不是抽象的彼此无关的信息零件。

比如说，从 score 数据表检索学生分数时，你想看的是考试日期而不是事件编号。这很容易办到：数据库将根据事件编号从 grade_event 表里查出考试日期来给你看。如果你还想知道考试分数是来自测验还是来自考试，这也很容易办到，数据库也能根据事件编号查出考分类型。别忘了，MySQL 之类的关系数据库系统最擅长的本领是——把一样东西与另一样东西联系起来，从多个信息源把你最想知道的信息查找出来。在考试记分的例子里，MySQL 必须通过事件编号才能对信息进行关联和提取，但你（数据库的使用者）并不需要关心这类细节。

为了让大家提前了解如何让 MySQL 将各信息联系起来，我们准备了一个例子，假设你打算查看 2002 年 9 月 23 日的考试分数。下面这个查询将那天的考试分数查出来：

```
SELECT score.name, grade_event.date, score.score, grade_event.category
FROM score INNER JOIN grade_event
ON score.event_id = grade_event.event_id
WHERE grade_event.date = '2008-09-23';
```

有点吓人吧？这个查询将把表 score 和表 event 结合（联系）起来并检索出学生姓名、考试日期、考试分数和考分类型等信息，下面是它的输出结果：

```
+--------+------------+-------+----------+
| name   | date       | score | category |
+--------+----------- +-------+----------+
| Billy  | 2008-09-23 |    15 | Q        |
| Missy  | 2008-09-23 |    14 | Q        |
| Johnny | 2008-09-23 |    17 | Q        |
| Jenny  | 2008-09-23 |    19 | Q        |
+--------+------------+-------+----------+
```

是不是觉得上面这个表格有点面熟？没错，它与图 1-4 里的表格格式是一模一样的。你不必知道事件编号就能得到这份查询结果。你指定了一个日期，MySQL 把该日期里的考试分数找了出来。总之，虽然数据库里的信息很抽象，与它们所代表的事物似乎也没有直接的联系，但这并不会影响它们的使用，数据库会根据你的查询把信息提取出来并显示为有意义的资料。

再仔细看看这个查询，大家也许又会产生一些新的问题。这个查询看起来太长太复杂。仅为查出某天的考试分数就要写这么多东西是不是太复杂了？是的。但是，有好几种办法能避免在输入查询命令的时候敲很多行内容。比较常见的做法是：一旦确定了某个查询的最终写法，就把它保存起来，以后在必要时就可以直接使用了。我们将在 1.5 节讨论这如何实现。

要不是为了让大家对查询过程有个了解，我是不想这么早就给出例子的。事实上，与我们真正用来检索考试分数的查询相比，刚才举的例子还算是简单的。因为我们还需要对数据表的布局再做一次较大的改动。首先，让 score 表不再包含学生姓名，我们将使用一个独一无二的学生 ID 编号。这其实就等于是用纸质记分簿里的 ID 栏而不是 Name 栏来构成 score 表。我们再新建一个名为 student

的数据表来存放学生姓名（name）和学号（student_id），如图 1-7 所示。

student数据表

name	sex	student_id
Billy	M	1
Missy	F	2
Johnny	M	3
Jenny	F	4

score数据表

student_id	event_id	score
1	5	15
2	5	14
3	5	17
4	5	19
1	6	67
2	6	73
3	6	82
4	6	79

grade_event数据表

event_id	date	category
1	2008-09-03	Q
2	2008-09-06	Q
3	2008-09-09	T
4	2008-09-16	Q
5	2008-09-23	Q
6	2008-10-01	T

图 1-7　通过学生学号和事件编号相联系的 score、student 和 grade_event 数据表

为什么要做这样的改动呢？为了应对出现两名学生名字相同的情况，唯一的学生 ID 将有助于把他们区分开来。（这与我们不使用日期而使用唯一事件编号来区分在同一天进行的考试和测试的分数的道理是一样的。）在对数据表的布局做了上述改动之后，用来查询给定日期的考试分数的命令又变得稍微复杂了一些，如下所示：

```
SELECT student.name, grade_event.date, score.score, grade_event.category
FROM grade_event INNER JOIN score INNER JOIN student
ON grade_event.event_id = score.event_id
AND score.student_id = student.student_id
WHERE grade_event.date = '2008-09-23';
```

如果你现在还看不懂这个查询命令，请不要着急。大多数初学者都是如此。在 1.4 节的后半部分内容里，我们还会遇到这个查询命令，等到那时你就能看明白它了。真的，不开玩笑。

大家可能已经注意到我在图 1-7 里的 student 表里增加了一些记分簿里没有的东西，它多了一个 sex（性别）列。你可以利用这个数据列来统计班级里男生或女生的人数，也可以利用它来对男女生的成绩进行比较分析。

考试记分项目的数据表到这里就设计得差不多了。我们只需再增加一个用来记录缺勤情况的数据表就全部完成了。这个数据表的内容很简单：一个学生 ID 和一个日期（如图 1-8 所示）。这个数据表里的每一个数据行都代表当天缺勤的一位学生。等到学期结束的时候，我们将通过 MySQL 的统计功能来汇总这个表里的数据，把每位学生的缺勤次数查出来。

absence 数据表

student_id	date
2	2008-09-02
4	2008-09-15
2	2008-09-20

图 1-8　absence 数据表

● student数据表

好了，考试记分项目的数据表到这里就全部设计完成了，下一步就该创建它们了。下面是我们用来创建 student 数据表的 CREATE TABLE 语句：

```
CREATE TABLE student
(
  name       VARCHAR(20) NOT NULL,
  sex        ENUM('F','M') NOT NULL,
  student_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (student_id)
) ENGINE = InnoDB;
```

注意观察，CREATE TABLE 语句中加入了一些新内容（结尾的 ENGINE 子句），稍后将解释它的

用途。

你可以在 mysql 客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_student.sql
```

上面这条 CREATE TABLE 语句将创建出一个名为的 student 且包含有 name、sex、student_id 等 3 个数据列的数据表。

name 是一个可变长度的字符串数据列，它最多可以容纳 20 个字符。这种人名表示法要比美国历史研究会场景中的数据表里使用多个数据列来分别保存人的姓氏和名字的情况来得简单，它只用了一个数据列。我之所以这样做是因为我知道考试记分项目不会出现必须用多个数据列来表示人名的查询操作。(因为这本书是我写的。但在实际中你可能需要使用多个数据列。)

sex 用来表明某位学生是男生还是女生。这是一个 ENUM（枚举）类型的数据列，其中的可取值只能是在该数据列的定义里枚举出来的那些值中的某一个：'F'代表女生，'M'代表男生。如果你想把某个数据列的可取值限制在一个元素个数有限的集合内，ENUM 就正好管用。当然了，我们也可以把这个数据列定义为 CHAR(1)，但 ENUM 能够更明确地把这个数据列只有有限个可取值的特点表示出来。如果你忘了它都有哪些可取值，可以发出一条 DESCRIBE 命令来查看。对于 ENUM 数据列，MySQL 将把它合法的枚举值都列出来，如下所示：

```
mysql> DESCRIBE student 'sex';
+-------+---------------+------+-----+---------+-------+
| Field | Type          | Null | Key | Default | Extra |
+-------+---------------+------+-----+---------+-------+
| sex   | enum('F','M') | NO   |     |         |       |
+-------+---------------+------+-----+---------+-------+
```

ENUM 数据列的值不一定非得是一个字符。我们完全可以把 sex 数据列定义为：ENUM（'female', 'male'）。

student_id 是一个整数类型的数据列，我们用它来保存学生的唯一编号。一般说来，学生们的学号应该从一个权威机构（如学校办公室）获得。但既然这只是一个示例性的数据表，我们不妨自己编造一些。我们使用了一个 AUTO_INCREMENT 数据列，对它的定义类似于在前面创建 member 数据表时对其 member_id 数据列的定义。

需要提醒大家的是，如果真的是从学校办公室获得学生 ID 而不是自动生成，就千万不要在定义 student_id 数据列时给它加上 AUTO_INCREMENT 属性。但为了避免出现重复的 ID 或 NULL ID 值，PRIMARY KEY 子句还是要保留下来的。

现在，CREATE TABLE 语句末尾的 ENGINE 子句是干什么用的？如果给出了这个子句，它将指定 MySQL 用来创建新数据表的存储引擎的名字。一种“存储引擎”就是一种用来管理某种特定类型的数据表的处理器。MySQL 有好几种存储引擎，每一种都有它自己的特性，我们将在 2.6.1 节对此展开讨论。

如果省略了 ENGINE 子句，MySQL 会替你选择一个默认引擎，它通常是 MyISAM。“ISAM”是“indexed sequential access method”（索引化顺序访问方法）的缩写，MyISAM 引擎在这种访问方法的基础上增加了一些 MySQL 独有的东西。因为我们刚才为“美国历史学会”创建数据表（president 和 member）的时候没有提供 ENGINE 子句，所以它们将是些 MyISAM 数据表（除非你曾重新配置过你的 MySQL 服务器，让其使用另外一种默认引擎）。至于那个考试成绩记录项目，我们明确地使用了

InnoDB存储引擎。InnoDB引擎通过引入“外键”概念而具备了保持“引用一致性”的特点。这意味着我们可以通过 MySQL 让数据表之间的关系满足一定的约束条件，而这对考试成绩记录项目中的数据表来说很有必要。

- ❑ 考试成绩与考试事件和学生是相关联的：如果某个学生ID和考试事件ID在student和grade_event数据表里尚不存在，就不应该把考试成绩录入到score数据表里去。
- ❑ 类似地，缺勤记录与学生相关联：如果某个学生ID在student数据表里尚不存在，就不应该把缺勤情况录入absence数据表。

为了满足这些条件，需要设置几个外键关系。“外”在这里的含义是“在另一个数据表里”，“外键”的含义是一个给定的键值必须与另一个数据表里的某个键值相匹配。这些概念会随着我们为考试成绩记录项目创建更多的数据表而变得越来越明晰。

- grade_event数据表

grade_event 数据表的定义如下所示：

```
CREATE TABLE grade_event
(
  date     DATE NOT NULL,
  category ENUM('T','Q') NOT NULL,
  event_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (event_id)
) ENGINE = InnoDB;
```

为了创建这个数据表，你既可以在mysql客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_grade_event.sql
```

date 数据列用来保存一个标准的 MySQL 日期值，必须写成'*CCYY-MM-DD*'的格式。

category 表示考试分数的类型。类似于 student 表里的 sex 列，category 也是一个枚举类型的数据列。它的可取值是'T'和'Q'，分别代表 test 和 quiz。

event_id 是一个 AUTO_INCREMENT 类型的数据列，并同时被定义为 PRIMARY KEY，它与 student 数据表里的 student_id 数据列情况类似。利用 AUTO_INCREMENT 属性，我们就能方便地生成唯一的事件编号了。与 student 数据表里的 student_id 数据列类似，这些编号到底是多少并不重要，重要的是它们必须是唯一的。

因为这些列中任何一个都不可以缺少值，所以将它们定义为 NOT NULL。

- score数据表

下面是用来创建 score 数据表的 CREATE TABLE 语句：

```
CREATE TABLE score
(
  student_id INT UNSIGNED NOT NULL,
  event_id   INT UNSIGNED NOT NULL,
  score      INT NOT NULL,
  PRIMARY KEY (event_id, student_id),
  INDEX (student_id),
  FOREIGN KEY (event_id) REFERENCES grade_event (event_id),
  FOREIGN KEY (student_id) REFERENCES student (student_id)
) ENGINE = InnoDB;
```

其中包含新内容：FOREIGN KEY 结构。稍后将介绍它。

为了创建这个数据表，既可以在 mysql 客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_score.sql
```

score 列是一个 INT，容纳整数值。如果想要纳入像 58.5 这样含小数部分的值，可以使用能表示它们的数据类型，如 DECIMAL 或 FLOAT。

student_id 和 event_id 数据列都是 INT（整数）类型的数据列，它们分别代表着每一个考试分数所对应的学生和考试事件。我们将通过它们把 student 和 grade_event 数据表联系起来以查出学生姓名和考试日期。student_id 和 event_id 数据列有一些需要注意的重点，如下所示。

- 我们已经把这两个数据列的组合设置为一个PRIMARY KEY。这确保了我们不会在某次考试或测验结束后重复录入某位学生的成绩。请注意，只有event_id和studemt_id的组合才具备我们需要的唯一性。在score数据表里，这两个ID值单独使用时都不具备唯一性的：同样的event_id值会出现在多个考试成绩数据行里（每位学生对应一个），同样的studemt_id值也会出现在多个数据行里（因为每位学生每参加一次考试或测验都会有一个成绩）。
- 每个ID数据列都需要一条FOREIGN KEY子句来定义它应该遵守的约束条件。这个子句的REFERENCES部分用来指定score数据列应该与哪个数据表里的哪个数据列相对应。event_id数据列上的约束条件是这个数据列里的每个值必须与grade_event数据表里的某个event_id值相匹配。类似地，score数据表里的每个student_id值必须与student数据表里的某个student_id值相匹配。

上面描述的 PRIMARY KEY 定义可以确保我们不会创建重复的考试成绩数据行，而 FOREIGN KEY 定义可以确保在我们的数据行不会有在 grade_event 或 student 数据表里并不存在的虚假 ID 值。

为什么 student_id 数据列上有一个索引？这是因为，对于出现在 FOREIGN KEY 定义里的每一个数据列，它要么本身有一个索引，要么是某个多数据列索引里第一个被列出的数据列。对于 event_id 数据列上的 FOREIGN KEY，该数据列在我们定义 PRIMARY KEY 时是第一个被列出来的。对于 student_id 数据列上的 FOREIGN KEY，就不能从 PRIMARY KEY 方面去找理由了，因为 student_id 数据列没被列在第一个。因此，我们需要在 student_id 数据列上另行创建一个索引。

值得一提的是，InnoDB 存储引擎其实会为出现在外键定义里的数据列自动创建一个索引，但它使用的索引定义不一定是你想要的（详见 2.14.1 节里的讨论），由你来明确地定义一个索引可以避免这个问题。

● absence数据表

下面是我们用来记录缺勤学生的 absence 数据表：

```
CREATE TABLE absence
(
  student_id INT UNSIGNED NOT NULL,
  date       DATE NOT NULL,
  PRIMARY KEY (student_id, date),
  FOREIGN KEY (student_id) REFERENCES student (student_id)
) ENGINE = InnoDB;
```

为了创建这个数据表，既可以在 mysql 客户程序里敲入上述语句，也可以在命令行上执行如下所示的命令：

```
% mysql sampdb < create_absence.sql
```

student_id 和 date 数据列都被定义为 NOT NULL，因为它们的内容都不允许缺失。为了避免这个数据表里出现重复的行，我决定把这两个数据列的组合也定义为一个主键。不管怎么说，把学生缺勤一天的情况统计为两次肯定是不公平的。

absence 表也包含一个外键关系，用来确保每个 student_id 值都与 student 表中的一个 student_id 值相匹配。

我们为考试成绩记录项目的数据表设置外键关系，是为了让那些约束条件能够在数据录入阶段发挥作用：只插入那些包含合法的考试事件 ID 值和学生 ID 值的数据行。不过，外键关系还有另外一种效果。它们会形成依赖关系，使你在创建和丢弃那些数据表的时候必须按照一定的顺序进行。

- score数据表依赖于grade_event和student数据表，所以在创建score数据表之前必须先创建出它们。类似地，adsence数据表依赖于student数据表，在创建adsence数据表时student数据表必须已经存在。
- 在丢弃数据表的时候，必须把上面的顺序倒过来。如果不先丢弃score数据表，就无法丢弃grade_event数据表；如果不先丢弃score和absence数据表，就无法丢弃student数据表。

注意 如果你的 MySQL 服务器因为某种原因不能提供 InnoDB 支持，你可以把考试成绩记录项目里的数据表创建为一些 MyISAM 数据表。把每一条 CREATE TABLE 语句里的 InnoDB 替换为 MyISAM 或者干脆省略 ENGINE 子句就能达到这一目的。不过，如果使用 MyISAM 数据表的话，本书后面的内容里用这些数据表去演示外键用法的例子就看不到效果了。

1.4.7 如何添加新的数据行

现在，我们已经把数据库和它里面的数据表都创建好了。接下来，我们需要往数据表里放一些行。但在此之前，我想先介绍一下如何查找数据表里的内容——在往里面放了一些记录之后，应该先看看自己做得怎么样吧。虽然我把有关检索操作的详细介绍安排在 1.4.9 节里，但你现在至少应该先把下面这条语句弄明白，它是用来查看名为 *tbl_name* 的数据表里的全部内容：

```
SELECT * FROM tbl_name;
```

例如：

```
mysql> SELECT * FROM student;
Empty set (0.00 sec)
```

现在，mysql 会报告说这个数据表是空的，但经过本小节中的几次示例操作之后，你就会看到不同的结果了。

往数据库添加数据的办法有好几种。你可以用 INSERT 语句以手工方式逐行插入到数据表里；也可以利用一个文件把行添加到数据表里，这个文件的内容既可以是一系列提前写好的 INSERT 语句（数据将通过客户程序 mysql 被加载到数据库里），也可以是纯粹的数据值（将通过 LOAD DATA 语句或 mysqlimport 工具程序被加载到数据库里）。

本小节将介绍把记录插入到数据表的各种方法。大家应该对它们都进行练习，熟悉并掌握它们的工作原理和用法。练习完这些方法之后，转到 1.4.8 节，运行其中的命令。这些命令用来删除数据表，然后重建，再把书中一整套已知数据加载到里面去。这样，你的数据库里的内容就与我将在后面示例中用到的数据一样了，而你自己做示例练习时看到的结果也将会与书中给出的结果一致。（如果已经

知道如何插入数据行，可以直接跳到本节末尾去填充你的数据表。）

1. 利用INSERT语句添加数据

我们先来学习如何用 INSERT 语句来添加数据记录。这是一条 SQL 语句，用来指定你打算往哪个数据表插入一个数据行以及该数据行的各数据列的值。INSERT 语句有好几种形式。

(1) **你可以一次性地列出全部数据列的值**，如下所示：

```
INSERT INTO tbl_name VALUES(value1,value2,...);
```

例如：

```
mysql> INSERT INTO student VALUES('Kyle','M',NULL);
mysql> INSERT INTO grade_event VALUES('2008-09-03','Q',NULL);
```

在使用这个语法的时候，关键字 VALUES 后面的括号里必须为数据表的全体数据列准备好对应的值，这些值的先后顺序也必须与各数据列在数据表里的存储先后顺序保持一致。（这个顺序通常就是各数据列在用来创建这个数据表的 CREATE TABLE 语句里的出现顺序。）如果你拿不准数据列的先后顺序，可以先用一条*数据表名称*语句来查一下。

MySQL 里的字符串或日期值必须放在单引号或双引号里才能被引用，放在单引号里更标准些。NULL 值对应于 student 和 event 数据表里的 AUTO_INCREMENT 数据列。在一个 AUTO_INCREMENT 数据列里插入一个表示“无数据”的 NULL 值将使 MySQL 为这个数据列自动生成下一个序号。

在 MySQL 中可以用一条 INSERT 语句把多个数据行插入到数据表里去，具体语法如下：

```
INSERT INTO tbl_name VALUES(...),(...),... ;
```

例如：

```
mysql> INSERT INTO student VALUES('Avery','F',NULL),('Nathan','M',NULL);
```

与刚才必须使用多条 INSERT 语句的情况相比，这种做法不仅能让你少打不少字，还能提高服务器的执行效率。注意，括号内包含了每行的一组列值。下列语句是非法的，因为它没在括号内包含正确数目的值。

```
mysql> INSERT INTO student VALUES('Avery','F',NULL,'Nathan','M',NULL);
ERROR 1136 (21S01): Column count doesn't match value count at row 1
```

(2) **还可以直接对数据列进行赋值，先给出数据列的名字，再列出它的值**。这特别适用于你创建的记录只有少数几个数据列需要有初始值的情况。具体语法如下：

```
INSERT INTO tbl_name (col_name1,col_name2,...) VALUES(value1,value2,...);
```

例如：

```
mysql> INSERT INTO member (last_name,first_name) VALUES('Stein','Waldo');
```

这种形式的 INSERT 语句一次可以插入多个记录：

```
mysql> INSERT INTO student (name,sex) VALUES('Abby','F'),('Joseph','M');
```

没有在 INSERT 语句中出现的数据列将被赋予默认值。例如，上面两条语句没有给出 member_id 或 event_id 数据列的值，所以 MySQL 将把默认值 NULL 赋给它们。（又因为 member_id 和 event_id 都是 AUTO_INCREMENT 数据列，所以结局将是这两个数据列被赋值为 MySQL 自动生成的下一个序列号。这与你直接把 NULL 赋值给它们的效果是一样的。）

(3) **还可以用包含 col_name = value（而非 VALUES()列表）的 SET 子句对数据列赋值。**

```
INSERT INTO tbl_name SET col_name1=value1, col_name2=value2, ... ;
```

例如：

```
mysql> INSERT INTO member SET last_name='Stein',first_name='Waldo';
```

没有在 SET 子句里出现的数据列将被赋予默认值。这种形式的 INSERT 语句不允许一次插入多个数据行。

既然知道了 INSERT 语句的工作原理，你可以用它去核实一下我们建立的外键关系是不是真的能够阻止“坏”数据行被录入到 score 和 absence 数据表。随便找几个没在 grade_event 或 student 数据表里出现过的 ID 值编造些“坏”数据行，看能不能把它们插入到数据表里：

```
mysql> INSERT INTO score (event_id,student_id,score) VALUES(9999,9999,0);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`sampdb`.`score`, CONSTRAINT `score_ibfk_1` FOREIGN
KEY (`event_id`) REFERENCES `grade_event` (`event_id`))
mysql> INSERT INTO absence SET student_id=9999, date='2008-09-16';
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`sampdb`.`absence`, CONSTRAINT `absence_ibfk_1`
FOREIGN KEY (`student_id`) REFERENCES `student` (`student_id`))
```

错误消息表明限制在起作用。

2. 通过从文件中读取来添加新行

把数据记录加载到数据表里的另一种方法从一个文件里把它们直接读出来。例如，如果 sampdb 发行版本里有一个名为 insert_president.sql 的文件，文件内容是一系列用来把新行添加到 president 数据表里的 INSERT 语句，你就可以像下面这样直接执行它们：

```
% mysql Sampdb < insert_president.sql
```

如果你已经进入 mysql，可以用一条 SOURCE 命令读入这个文件，如下所示：

```
mysql> source insert_president.sql;
```

SOURCE 命令只能用在 MySQL 3.23.9 或更高的版本里。

如果文件里的记录项不是以 INSERT 语句而是以纯数据值的形式来存放的，我们可以利用 LOAD DATA 语句或 mysqlimport 工具程序来加载它们。

LOAD DATA 语句就像是一架大型装载机，它能把文件里的数据一次性地全部读到数据表里。这条语句要在 mysql 客户程序里使用：

```
mysql> LOAD DATA LOCAL INFILE 'member.txt' INTO TABLE member;
```

假设数据文件 member.txt 就保存在你正使用着的客户主机上的当前子目录里，上面这条语句将读这个文件并把它的内容发送到服务器以加载到 member 数据表。（你可以在 sampdb 发行版本里找到这个 member.txt 文件。）

在默认的情况下，LOAD DATA 语句将假设各数据列的值以制表符分隔，各数据行以换行符分隔，数据值的排列顺序与各数据列在数据表里的先后顺序一致。但你完全可以用它来读取其他格式的数据文件或者按其他顺序来读取各数据列的值，有关细节请参阅附录 E 里的 LOAD DATA 条目。

LOAD DATA 中的 LOCAL 关键字可以使客户程序（本例中是 mysql）读取数据文件并发送到服务器以加载。如果省略了关键字 LOCAL，就表示数据文件是保存在服务器主机上的，而你必须拥有相应的 FILE 服务器访问权限才能把文件里的数据加载到数据表里去。但可惜的是，大多数 MySQL 用户都没

有这种权限。还应指定文件的完全路径，这样服务器才能找到它。

如果使用 LOAD DATA LOCAL 时得到下面的错误，LOCAL 功能在默认的情况下就可能处于禁用状态。

```
ERROR 1148 (42000): The used command is not allowed with this MySQL version
```

你可以试试加上--local-infile 选项再重新启动一次 mysql 程序的办法，比如：

```
% mysql --local-infile sampdb
mysql> LOAD DATA LOCAL INFILE 'member.txt' INTO TABLE member;
```

如果这一招也不管用，就说明服务器端的 LOCAL 机制没有被激活。激活服务器端 LOCAL 机制的具体做法请参阅第 12 章。

加载数据文件的另一种方法是使用 mysqlimport 客户程序。当你在命令提示符下启动 mysqlimport 程序时，它会为你生成一条 LOAD DATA 语句：

```
% mysqlimport --local sampdb member.txt
```

此外，与你使用 mysql 客户程序时一样，如果还需要设定连接参数，请在命令行上把它们添加到数据库名称的前面。

就上面这条命令而言，mysqlimport 程序将生成一条能够把 member.txt 文件里的数据值加载到 member 数据表里去的 LOAD DATA 语句。这是因为 mysqlimport 程序是根据数据文件的名字来确定与之对应的数据表的，它将把文件名中第一个句号字符（.）之前的那个字符串用做数据表的名字。举例来说，它会把 member.txt 和 president.txt 文件里的数据分别加载到 member 和 president 数据表里去。这就要求你必须慎重选择数据文件的名字，要不然，mysqlimport 程序就会把数据错误地加载到别的数据表里去。我们来看一个例子：如果你想把 member1.txt 和 member2.txt 文件里的数据都加载到 member 数据表里去，但 mysqlimport 却会认为你想把这两个文件分别加载到名为 member1 和 member2 的两个数据表里去。为了避免出现这种混乱，你可以把这两个文件命名为 member.1.txt 和 member.2.txt，或者是 member.txt1 和 member.txt2。

1.4.8　将sampdb数据库重设为原来的状态

在练习完上面介绍的这几种数据行添加方法之后，为了顺利进行后面的学习，你应该重新创建和加载 sampdb 数据库里的数据表，把它们的内容恢复为原样。使用包含 sampdb 发布文件的目录中的 mysql 程序，写出以下语句：

```
% mysql sampdb
mysql> source create_member.sql;
mysql> source create_president.sql;
mysql> source insert_member.sql;
mysql> source insert_president.sql;
mysql> DROP TABLE IF EXISTS absence, score, grade_event, student;
mysql> source create_student.sql;
mysql> source create_grade_event.sql;
mysql> source create_score.sql;
mysql> source create_absence.sql;
mysql> source insert_student.sql;
mysql> source insert_grade_event.sql;
mysql> source insert_score.sql;
```

```
mysql> source insert_absence.sql;
```

如果你不喜欢输入这么多条命令，那么，在 Unix 系统上，请执行下面这条命令：

```
% sh init_all_tables.sh sampdb
```

在 Windows 系统上，请执行下面这条命令：

```
C:\> init_all_tables.bat sampdb
```

无论使用哪条命令，如果你还需要设定连接参数，请在命令行上把它们添加到命令名称的后面。

1.4.9 检索信息

现在，数据表都已经创建出来并加载了数据。下面看看这些数据都能派上哪些用场。SELECT 语句允许以你喜欢的方式检索和显示数据表里的信息。例如，可以像下面这样把整个数据表的内容都显示出来：

```
SELECT * FROM president;
```

也可以像下面这样只选取某个数据行里的某个数据列：

```
SELECT birth FROM president WHERE last_name = 'Eisenhower';
```

SELECT 语句还有好几个子句（也叫组成部分），它们的各种搭配能帮你查出你最感兴趣的信息。这些子句可以很简单，也可以很复杂，由它们搭配出来的 SELECT 语句也会相应地变得简单或者复杂。不过，请放心，在本书里，绝不会有长达数页让大家必须花费好几个钟头才能搞明白的查询命令。（如果在看书时遇到长长的查询命令，我通常会跳过它们，我想你也会如此。）

下面是 SELECT 语句的通用形式：

```
SELECT what to retrieve
FROM table or tables
WHERE conditions that data must satisfy;
```

在写 SELECT 语句时，先把你想检索的东西说清楚，再把可选子句写出来。上面两个子句（FROM 和 WHERE）是最常见的，其他子句包括 GROUP BY、ORDER BY 和 LIMIT 等。需要指出的是，SQL 语言对书写格式并没有严格的要求，所以在书写 SELECT 语句时，放置换行符的位置不必与本书示例中的一样。

FROM 子句一般都少不了，但如果你不需要给出数据表的名字，就不必把它写出来。比如说，下面这条语句只是计算一个表达式的值。因为这个计算不涉及任何数据表，所以没有必要把 FROM 子句写出来：

```
mysql> SELECT 2+2, 'Hello, world', VERSION();
+-----+--------------+------------+
| 2+2 | Hello, world | VERSION()  |
+-----+--------------+------------+
|   4 | Hello, world | 5.0.60-log |
+-----+--------------+------------+
```

当的确需要使用一个 FROM 子句来指定将从哪个数据表检索数据时，你还需要把想查看的数据列的名字列举出来。SELECT 语句最常用形式是用一个星号（表示所有数据列）作为数据列说明符。下面这条查询将把 student 数据表所有的数据列全都显示出来：

```
mysql> SELECT * FROM student;
+-----------+-----+------------+
```

```
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Kyle      | M   |          3 |
| Katie     | F   |          4 |
...
```

有关数据列将按它们在数据表里的存储先后顺序显示出来。这个顺序与你用 DESCRIBE student 语句查看到的数据列排列顺序是一致的。(示例末尾处的省略号表示这个查询所返回的数据行比大家在这里看到的要多。)

你也可以把自己想要查看的数据列的名字逐个列出来。例如，如果你只想查看学生姓名，就应该使用下面这条语句：

```
mysql> SELECT name FROM student;
+-----------+
| name      |
+-----------+
| Megan     |
| Joseph    |
| Kyle      |
| Katie     |
...
```

如果需要列举多个数据列，请用逗号把它们分隔开。请看下面这条语句，它与 SELECT* FROM student 是等价的，但它把各数据列的名字都明确地列了出来：

```
mysql> SELECT name, sex, student_id FROM student;
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Kyle      | M   |          3 |
| Katie     | F   |          4 |
...
```

你可以按任意顺序列举数据列的名字：

```
SELECT name, student_id FROM student;
SELECT student_id, name FROM student;
```

只要你愿意，甚至还可以重复列举数据列的名字，只是这样做通常没有多大的意义。

MySQL 允许你在一条 SELECT 语句里同时选取多个数据表里的数据列，这叫做数据表之间的联结(join)，我们将在第 4 小节里对此进行讨论。

MySQL 里的数据列名称不区分字母的大小写，所以下面这些查询都是等价的：

```
SELECT name, student_id FROM student;
SELECT NAME, STUDENT_ID FROM student;
SELECT nAmE, sTuDeNt_Id FROM student;
```

但需要注意的是，数据库和数据表的名字却可能需要区分字母的大小写，这取决于服务器主机上所使用的文件系统，以及 MySQL 的配置情况。比如说，Windows 文件名不区分大小写，所以运行在 Windows 系统上的服务器也就不区分数据库和数据表名字的大小写；Unix 文件名区分大小写，所以运

行在 Unix 系统上的服务器就将区分数据库和数据表名字的大小写。（属于 Unix 阵营的 Mac OS X 是个例外：HFS+文件系统不区分文件名的大小写，但 UFS 文件系统却区分。）如果你想让 MySQL 服务器不区分数据库和数据表名字中的大小写，可以对服务器进行配置，请参阅 11.2.5 节。

1. 指定检索条件

要想让 SELECT 语句只把满足特定条件的记录检索出来，就必须给它加上一个 WHERE 子句来设定数据行的检索条件。只有这样，你才能有选择地把数据列的取值满足特定要求的那些数据行挑选出来。你可以针对任何类型的值进行查找。例如，可以针对某些数值进行搜索：

```
mysql> SELECT * FROM score WHERE score > 95;
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          5 |        3 |    97 |
|         18 |        3 |    96 |
|          1 |        6 |   100 |
|          5 |        6 |    97 |
|         11 |        6 |    98 |
|         16 |        6 |    98 |
+------------+----------+-------+
```

也可以针对字符串值进行查找。对于默认的字符设置和排序，字符串的比较操作通常不区分字母的大小写：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='ROOSEVELT';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='roosevelt';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
```

还可以针对日期值进行查找：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < '1750-1-1';
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
+------------+------------+------------+
```

甚至还能针对不同类型的值的组合进行查找：

```
mysql> SELECT last_name, first_name, birth, state FROM president
    -> WHERE birth < '1750-1-1' AND (state='VA' OR state='MA');
+------------+------------+------------+-------+
| last_name  | first_name | birth      | state |
+------------+------------+------------+-------+
| Washington | George     | 1732-02-22 | VA    |
| Adams      | John       | 1735-10-30 | MA    |
| Jefferson  | Thomas     | 1743-04-13 | VA    |
+------------+------------+------------+-------+
```

WHERE 子句里的表达式允许使用算术运算符（见表1-1）、比较运算符（见表1-2）和逻辑运算符（见表1-3），还允许使用括号。你可以使用常数、数据表的数据列和函数调用进行运算。在这本书的查询示例里用到的 MySQL 函数不算很多，但它们的数量其实并不少。MySQL 函数的完整清单请参阅附录 C。

表1-1　算术运算符

运算符	含　义
+	加法
-	减法
*	乘法
/	除法
DIV	整数除法
%	求余（整数除法后的剩余部分）

表1-2　比较运算符

运算符	含　义
<	小于
<=	小于或等于（不大于）
=	等于
<=>	等于（能够对NULL值进行比较）
<> 或 !=	不等于
>=	大于或等于（不小于）
>	大于

表1-3　逻辑运算符

运算符	含　义
AND	逻辑与
OR	逻辑或
XOR	逻辑异或
NOT	逻辑非

如果需要在查询语句里使用逻辑运算符，千万要注意一点：逻辑 AND 运算与人们日常生活中所说的“和”在含义上是不一样的。举个例子，假设你想把“出生于弗吉尼亚州和出生于马萨诸塞州的总统”查出来。这道问题里有一个“和”字，头脑简单的人会不假思索地写出一条下面这样的查询命令：

```
mysql> SELECT last_name, first_name, state FROM president
    -> WHERE state='VA' AND state='MA';
Empty set (0.36 sec)
```

这个查询的结果集显然是空的，没有把我们想要的东西找出来。为什么会这样呢？因为这条查询的真正含义是“把同时出生在弗吉尼亚州和马萨诸塞州的总统”找出来，而这种情况是不可能出现的。在日常生活里，你可以用“和”来表达你的查询条件，但在 SQL 里，你却必须把这两个条件用逻辑或操作符 OR 并列在一起，即：

```
mysql> SELECT last_name, first_name, state FROM president
    -> WHERE state='VA' OR state='MA';
+------------+-------------+-------+
| last_name  | first_name  | state |
+------------+-------------+-------+
| Washington | George      | VA    |
| Adams      | John        | MA    |
| Jefferson  | Thomas      | VA    |
| Madison    | James       | VA    |
| Monroe     | James       | VA    |
| Adams      | John Quincy | MA    |
```

```
| Harrison   | William H.  | VA    |
| Tyler      | John        | VA    |
| Taylor     | Zachary     | VA    |
| Wilson     | Woodrow     | VA    |
| Kennedy    | John F.     | MA    |
| Bush       | George H.W. | MA    |
+------------+-------------+-------+
```

请大家务必注意日常语言与 SQL 语句之间的这类差异——在为你自己编写查询命令时要注意，在为其他人编写查询命令时更要注意。一定要仔细倾听别人对查询内容的描述，然后根据描述正确地选用适当的 SQL 逻辑操作符。以刚才那个查询为例，正确的自然语言表述形式应该是："把出生于弗吉尼亚州或者出生于马萨诸塞州的总统给找出来。"

你可能会发现像下面这样，在表达查询时，用 IN()操作符来查找几个值中的某一个会很方便。前面的查询可以使用 IN()写成下面这样：

```
SELECT last_name, first_name, state FROM president
WHERE state IN('VA','MA');
```

在比较一个列和一大组值时，IN()使用起来特别方便。

2. NULL值

NULL 是一个很特殊的值。它的含义是"无数据"或"未知数据"，所以不能用它与"有数据"的值进行运算或者比较。如果你试图用普通的算术比较运算符对 NULL 值进行操作，其结果将是不可预料的：

```
mysql> SELECT NULL < 0, NULL = 0, NULL <> 0, NULL > 0;
+----------+----------+-----------+----------+
| NULL < 0 | NULL = 0 | NULL <> 0 | NULL > 0 |
+----------+----------+-----------+----------+
|     NULL |     NULL |      NULL |     NULL |
+----------+----------+-----------+----------+
```

事实上，你甚至不应该把 NULL 值与它本身进行比较，因为两个表示"无数据"的未知值的比较结果也将是不可预料的：

```
mysql> SELECT NULL = NULL, NULL <> NULL;
+-------------+--------------+
| NULL = NULL | NULL <> NULL |
+-------------+--------------+
|        NULL |         NULL |
+-------------+--------------+
```

如果需要对 NULL 值进行查找，就必须使用一种特殊的语法。你不能使用=、<>或者!=来测试它们是相等还是不相等，你必须使用 IS NULL 或 IS NOT NULL 来判断。例如，如果你想把目前仍然健在的美国总统给查出来，就应该使用一条下面这样的查询命令，因为这些总统的逝世日期在 president 数据表里是用 NULL 值来表示的：

```
mysql> SELECT last_name, first_name FROM president WHERE death IS NULL;
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Carter    | James E.    |
| Bush      | George H.W. |
```

```
| Clinton   | William J.  |
| Bush      | George W.   |
+-----------+-------------+
```

如果想把没有姓名后缀的美国总统查出来，就应该在检索条件里使用 IS NOT NULL 进行判断：

```
mysql> SELECT last_name, first_name, suffix
    -> FROM president WHERE suffix IS NOT NULL;
+-----------+------------+--------+
| last_name | first_name | suffix |
+-----------+------------+--------+
| Carter    | James E.   | Jr.    |
+-----------+------------+--------+
```

专用的 MySQL 比较操作符<=>能完成 NULL 值与 NULL 值之间的比较。你可以用这个操作符把刚才的两条查询命令分别改写为：

```
SELECT last_name, first_name FROM president WHERE death <=> NULL;

SELECT last_name, first_name, suffix
FROM president WHERE NOT (suffix <=> NULL);
```

3. 如何对查询结果进行排序

只要你是 MySQL 用户，迟早会注意到这样一种情况：如果你创建了一个数据表并往里面加载了一些数据记录，当你发出一条“SELECT * FROM 数据表名称”语句时，数据记录在查询结果中的先后顺序通常与它们当初被插入时的先后顺序一致。这很符合人们的思维习惯，人们很自然地假设数据记录在查询结果中的先后顺序与它们当初被插入时的先后顺序相同。但这是不正确的，因为如果你在加载完数据表的初始数据之后又删除并插入了一些数据行，这些操作往往会改变数据行在服务器所返回的数据表检索结果中的先后顺序。（数据删除操作会在数据表里留下一些“空洞”，而 MySQL 会用你以后插入的新记录来尽量填补这些“空洞”。）

你真正可以信赖的原则是：从服务器返回的数据行的先后顺序没有任何保证，除非你事先设定。如果想让查询结果按你希望的先后顺序显示，就必须给查询命令增加一条 ORDER BY 子句。下面这条查询命令将把美国总统们的姓名按姓氏字母表顺序排列并显示出来：

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY last_name;
+------------+---------------+
| last_name  | first_name    |
+------------+---------------+
| Adams      | John          |
| Adams      | John Quincy   |
| Arthur     | Chester A.    |
| Buchanan   | James         |
...
```

ORDER BY 子句中的默认排序方式是升序排列。在 ORDER BY 子句中的数据列名字的后面加上关键字 ASC 或 DESC，就能使查询结果中的数据记录按指定数据列的升序或者降序排列，例如，如果你想让美国总统们的姓名按姓氏的逆序（降序）排列显示，就应该像下面这样加上一个 DESC 关键字：

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY last_name DESC;
```

```
+------------+---------------+
| last_name  | first_name    |
+------------+---------------+
| Wilson     | Woodrow       |
| Washington | George        |
| Van Buren  | Martin        |
| Tyler      | John          |
...
```

可以对查询结果按多个数据列进行排序，而每一个数据列又都可以互不影响地分别按升序或降序进行排列。下面这条针对 president 数据表的查询命令将把查询结果中的数据行按总统出生地所在州的逆序排列，而出生地所在州相同的总统姓名又将按其姓氏的升序排列：

```
mysql> SELECT last_name, first_name, state FROM president
    -> ORDER BY state DESC, last_name ASC;
+------------+---------------+-------+
| last_name  | first_name    | state |
+------------+---------------+-------+
| Arthur     | Chester A.    | VT    |
| Coolidge   | Calvin        | VT    |
| Harrison   | William H.    | VA    |
| Jefferson  | Thomas        | VA    |
| Madison    | James         | VA    |
| Monroe     | James         | VA    |
| Taylor     | Zachary       | VA    |
| Tyler      | John          | VA    |
| Washington | George        | VA    |
| Wilson     | Woodrow       | VA    |
| Eisenhower | Dwight D.     | TX    |
| Johnson    | Lyndon B.     | TX    |
...
```

对于包含 NULL 值的数据行，如果设定按升序排列，它们将出现在查询结果的开头；如果设定按降序排列，它们将出现在查询结果的末尾。如果你想让包含 NULL 值的数据行必须出现在查询结果的末尾，就必须额外增加一个排序数据列以区分 NULL 值和非 NULL 值。例如，如果你想按逝世日期的降序对总统们的姓名排序，那么健在（即逝世日期等于 NULL）的总统们的姓名将出现在查询结果的末尾，而如果想让后者出现在查询结果的开头，就应该使用一条下面这样的查询命令：

```
mysql> SELECT last_name, first_name, death FROM president
    -> ORDER BY IF(death IS NULL,0,1), death DESC;
+------------+---------------+------------+
| last_name  | first_name    | death      |
+------------+---------------+------------+
| Clinton    | William J.    | NULL       |
| Bush       | George H.W.   | NULL       |
| Carter     | James E.      | NULL       |
| Bush       | George W.     | NULL       |
| Ford       | Gerald R.     | 2006-12-26 |
| Reagan     | Ronald W.     | 2004-06-05 |
| Nixon      | Richard M.    | 1994-04-22 |
| Johnson    | Lyndon B.     | 1973-01-22 |
...
| Jefferson  | Thomas        | 1826-07-04 |
```

```
| Adams      | John          | 1826-07-04 |
| Washington | George        | 1799-12-14 |
+------------+---------------+------------+
```

IF()函数的作用是对紧随其后的表达式（第一参数）进行求值，再根据表达式求值结果的真假返回它的第二参数或第三参数。在刚才的第一个查询里，在遇到 NULL 值的时候，IF()函数的求值结果将是 0；在遇到非 NULL 值的时候，IF()函数的求值结果将是 1。最终结果是把包含 NULL 值的数据行全都放在了没有 NULL 值的数据行的前面。

4. 如何限制查询结果中的数据行个数

查询结果往往由很多个数据行构成，如果你只想看到其中的一小部分，可以给查询命令增加一条 LIMIT 子句，它与 ORDER BY 子句联合使用的效果往往更佳。MySQL 允许给查询结果中的数据行个数设置一个上限，如前 *n* 个数据行，如果查询结果里的数据行超过了这个数字，就只显示前 *n* 个数据行。下面这个查询将把出生日期最早的前 5 个总统列举出来：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth LIMIT 5;
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

如果你使用了 ORDER BY birth DESC（即按逆序）来排序查询结果，你找出来的就将是出生日期最晚的 5 个总统：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth DESC LIMIT 5;
+-----------+-------------+------------+
| last_name | first_name  | birth      |
+-----------+-------------+------------+
| Clinton   | William J.  | 1946-08-19 |
| Bush      | George W.   | 1946-07-06 |
| Carter    | James E.    | 1924-10-01 |
| Bush      | George H.W. | 1924-06-12 |
| Kennedy   | John F.     | 1917-05-29 |
+-----------+-------------+------------+
```

LIMIT 还允许从查询结果的中间部分抽出一部分数据记录。此时必须设定两个值，第一个值给出了要在查询结果的开头部分跳过的数据记录个数，第二个值则是需要返回的数据记录的个数。下面的查询与前一个很相似，但它返回的是跳过前 10 个数据记录之后的 5 个数据记录：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> ORDER BY birth DESC LIMIT 10, 5;
+-----------+-------------+------------+
| last_name | first_name  | birth      |
+-----------+-------------+------------+
| Truman    | Harry S     | 1884-05-08 |
| Roosevelt | Franklin D. | 1882-01-30 |
```

```
| Hoover     | Herbert C.  | 1874-08-10 |
| Coolidge   | Calvin      | 1872-07-04 |
| Harding    | Warren G.   | 1865-11-02 |
+------------+-------------+------------+
```

如果想从president数据表里随机抽取出一条或一组数据记录，可以联合使用LIMIT和ORDER BY RAND()子句，如下所示：

```
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY RAND() LIMIT 1;
+-----------+------------+
| last_name | first_name |
+-----------+------------+
| Johnson   | Lyndon B.  |
+-----------+------------+
mysql> SELECT last_name, first_name FROM president
    -> ORDER BY RAND() LIMIT 3;
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Harding   | Warren G.   |
| Bush      | George H.W. |
| Jefferson | Thomas      |
+-----------+-------------+
```

5. 如何对输出列进行求值和命名

前面给出的查询示例的输出结果大都是直接检索自数据表的数据值。MySQL 还允许把表达式的计算结果当做输出列的值，而不引用数据表。表达式可以很简单，也可以很复杂。就拿下面这个查询来说吧，它有两个输出列，前一个输出列对应着一个非常简单的表达式（一个常数），而后一个输出列则对应着一个使用了多个算术运算符和多个函数调用的复杂表达式，它生成一个平方根，并将结果按 3 位小数位表示：

```
mysql> SELECT 17, FORMAT(SQRT(25+13),3);
+----+-----------------------+
| 17 | FORMAT(SQRT(25+13),3) |
+----+-----------------------+
| 17 | 6.164                 |
+----+-----------------------+
```

数据表里的数据列名字也可以用在表达式里，如下所示：

```
mysql> SELECT CONCAT(first_name,' ',last_name),CONCAT(city,', ',state)
    -> FROM president;
+----------------------------------+-------------------------+
| CONCAT(first_name,' ',last_name) | CONCAT(city,', ',state) |
+----------------------------------+-------------------------+
| George Washington                | Wakefield, VA           |
| John Adams                       | Braintree, MA           |
| Thomas Jefferson                 | Albemarle County, VA    |
| James Madison                    | Port Conway, VA         |
...
```

在这个查询里，我们对输出列的格式进行了设置：总统们的名字和姓氏合二为一，成为了一个以空格分隔的字符串；他们的出生城市和出生州则被合并为一个以逗号分隔的字符串。

如果输出列的值是某个表达式的结算结果，这个表达式就会成为这个输出列的名字并用作它在输出结果中的标题。如果表达式很长（例如上面这个查询示例），就会使输出列的宽度变得很大。为解决这一问题，你可以利用 AS name 短语给输出列另外取一个名字，我们把它们称为（输出）列的别名(alias)。例如，我们可以像下面这样把上面这个查询的输出结果改写得更有意义：

```
mysql> SELECT CONCAT(first_name,' ',last_name) AS Name,
    -> CONCAT(city,', ',state) AS Birthplace
    -> FROM president;
+-----------------------+------------------------+
| Name                  | Birthplace             |
+-----------------------+------------------------+
| George Washington     | Wakefield, VA          |
| John Adams            | Braintree, MA          |
| Thomas Jefferson      | Albemarle County, VA   |
| James Madison         | Port Conway, VA        |
...
```

如果输出列的别名里包含空格符，就必须把它放到一组引号里，如下所示：

```
mysql> SELECT CONCAT(first_name,' ',last_name) AS 'President Name',
    -> CONCAT(city,', ',state) AS 'Place of Birth'
    -> FROM president;
+-----------------------+------------------------+
| President Name        | Place of Birth         |
+-----------------------+------------------------+
| George Washington     | Wakefield, VA          |
| John Adams            | Braintree, MA          |
| Thomas Jefferson      | Albemarle County, VA   |
| James Madison         | Port Conway, VA        |
...
```

在为数据列提供别名时，关键字 AS 可以省略：

```
mysql> SELECT 1, 2 AS two, 3 three;
+---+-----+-------+
| 1 | two | three |
+---+-----+-------+
| 1 |   2 |     3 |
+---+-----+-------+
```

我个人更喜欢写出 AS。如果省略了它，稍不留神就会写出一个看起来完全合法但执行起来却结果迥异的查询命令来。比如说，你本打算编写一个查询命令去选择总统的姓名，只可惜你不小心漏掉了 first_name 和 last_name 数据列之间的逗号，写出了下面这样的语句：

```
mysql> SELECT first_name last_name FROM president;
+---------------+
| last_name     |
+---------------+
| George        |
| John          |
| Thomas        |
| James         |
...
```

这样一来，查询结果不再是两个输出列了。它只能显示 first_name 数据列的内容，last_name

被视为一个数据列别名而成为了输出列的表头。如果某个查询命令检索出来的输出列的个数不符合你的预期，并且输出列的名字也不符合你的预期，就应该去检查一下是不是在什么地方漏掉了数据列之间的逗号。

6. 与日期有关的问题

在与 MySQL 里的日期打交道的时候，千万要记住年份总是出现在最前面。2008 年 7 月 27 日将被表示为'2002-07-27'，而不是像在日常生活中那样被表示为'07-27-2002'或'27-07-2002'。

MySQL 已经为我们准备了一些日期操作，比较常见的有下面几种。

- 按日期排序。（我们已经见过几个这方面的例子了。）
- 查找某个日期或者某个日期范围。
- 提取日期值中的年、月、日等组成部分。
- 计算两个日期之间的时间距离。
- 用一个日期加上或减去一个时间间隔以求出另一个日期。

下面是一些与日期有关的查询示例。

如果你的查询与某特定日期有关，就得把一个 DATE 类型的数据列与你感兴趣的那个日期值进行比较，如下所示：

```
mysql> SELECT * FROM grade_event WHERE date = '2008-10-01';
+------------+----------+----------+
| date       | category | event_id |
+------------+----------+----------+
| 2008-10-01 | T        |        6 |
+------------+----------+----------+
mysql> SELECT last_name, first_name, death
    -> FROM president
    -> WHERE death >= '1970-01-01' AND death < '1980-01-01';
+-----------+------------+------------+
| last_name | first_name | death      |
+-----------+------------+------------+
| Truman    | Harry S    | 1972-12-26 |
| Johnson   | Lyndon B.  | 1973-01-22 |
+-----------+------------+------------+
```

日期中的年、月、日 3 部分可以用函数 YEAR()、MONTH()、DAYOFMONTH()分别分离出来。例如，下面这个查询将把生日在 3 月的美国总统查出来：

```
mysql> SELECT last_name, first_name, birth
    -> FROM president WHERE MONTH(birth) = 3;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Madison   | James      | 1751-03-16 |
| Jackson   | Andrew     | 1767-03-15 |
| Tyler     | John       | 1790-03-29 |
| Cleveland | Grover     | 1837-03-18 |
+-----------+------------+------------+
```

在这个查询里，还可以直接使用 3 月份的英文名称 March：

```
mysql> SELECT last_name, first_name, birth
```

```
    -> FROM president WHERE MONTHNAME(birth) = 'March';
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Madison   | James      | 1751-03-16 |
| Jackson   | Andrew     | 1767-03-15 |
| Tyler     | John       | 1790-03-29 |
| Cleveland | Grover     | 1837-03-18 |
+-----------+------------+------------+
```

再进一步，把 MONTH()和 DAYOFMONTH()函数结合起来使用，就能把生日是 3 月的某一天的总统查出来：

```
mysql> SELECT last_name, first_name, birth
    -> FROM president WHERE MONTH(birth) = 3 AND DAYOFMONTH(birth) = 29;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Tyler     | John       | 1790-03-29 |
+-----------+------------+------------+
```

很多报纸的娱乐版都有“今日出生的名人”之类的栏目，而上面的查询就能生成一份这样的名单。不过，如果你的查询与“今天”有关，那大可不必像前面的例子那样使用一个具体的日期值。MySQL 为我们准备了一个 CURDATE()函数，它的返回值永远是“今天”的日期值。于是，无论“今日”是哪一天，将总统的生日与 CURDATE()比较，“今日出生的总统”就可以用下面这个查询找出来：

```
SELECT last_name, first_name, birth
FROM president WHERE MONTH(birth) = MONTH(CURDATE())
AND DAYOFMONTH(birth) = DAYOFMONTH(CURDATE());
```

如果你想知道两个日期值之间的时间间隔，拿它们做减法就行了。例如，如果你想知道哪位总统的寿命最长，就得用他们的逝世日期减去他们的出生日期。此时 TIMESTAMPDIFF()函数很有用，有接受一个参数，用以指定计算结果的单位，本例中是年，即 YEAR。如下所示：

```
mysql> SELECT last_name, first_name, birth, death,
    -> TIMESTAMPDIFF(YEAR, birth, death) AS age
    -> FROM president WHERE death IS NOT NULL
    -> ORDER BY age DESC LIMIT 5;
+-----------+------------+------------+------------+------+
| last_name | first_name | birth      | death      | age  |
+-----------+------------+------------+------------+------+
| Reagan    | Ronald W.  | 1911-02-06 | 2004-06-05 |   93 |
| Ford      | Gerald R.  | 1913-07-14 | 2006-12-26 |   93 |
| Adams     | John       | 1735-10-30 | 1826-07-04 |   90 |
| Hoover    | Herbert C. | 1874-08-10 | 1964-10-20 |   90 |
| Truman    | Harry S    | 1884-05-08 | 1972-12-26 |   88 |
+-----------+------------+------------+------------+------+
```

计算日期间隔的另一种方法是使用 TO_DAYS 函数，它将日期都转换为天数。这个函数可以计算出距某特定日期还有多长的时间。例如，可以这样找出需要在近期交纳会费的美国历史研究会会员：用会员资格的失效日期减去“今天”的日期，若其结果小于某个极值，就表明这位会员需要续费了。下面这个查询将把资格已经失效和需要在 60 天以内续交会费的会员查出来：

```
SELECT last_name, first_name, expiration FROM member
WHERE (TO_DAYS(expiration) - TO_DAYS(CURDATE())) < 60;
```

使用 TIMESTAMPDIFF()函数也可以完成，如下所示：

```
SELECT last_name, first_name, expiration FROM member
WHERE TIMESTAMPDIFF(DAY, CURDATE(), expiration) < 60;
```

日期值的用 DATE_ADD()或 DATE_SUB()函数来完成。这两个函数的输入参数是一个日期值和一个时间间隔值，返回结果则是一个新日期值。请看下面的例子：

```
mysql> SELECT DATE_ADD('1970-1-1', INTERVAL 10 YEAR);
+----------------------------------------+
| DATE_ADD('1970-1-1', INTERVAL 10 YEAR) |
+----------------------------------------+
| 1980-01-01                             |
+----------------------------------------+
mysql> SELECT DATE_SUB('1970-1-1', INTERVAL 10 YEAR);
+----------------------------------------+
| DATE_SUB('1970-1-1', INTERVAL 10 YEAR) |
+----------------------------------------+
| 1960-01-01                             |
+----------------------------------------+
```

在本小节的前半部分有一个用来查询“哪些美国总统逝世于 20 世纪 70 年代”的示例，里面用了两个确切的日期值来表示时间的起止点。利用日期值的加减法运算，原来的查询可以被改写为：起点日期仍使用一个确切的日期，但终点日期却是通过起点日期加上一个时间间隔而计算出来的。如下所示：

```
mysql> SELECT last_name, first_name, death
    -> FROM president
    -> WHERE death >= '1970-1-1'
    -> AND death < DATE_ADD('1970-1-1', INTERVAL 10 YEAR);
+-----------+------------+------------+
| last_name | first_name | death      |
+-----------+------------+------------+
| Truman    | Harry S    | 1972-12-26 |
| Johnson   | Lyndon B.  | 1973-01-22 |
+-----------+------------+------------+
```

用来查找“需要在近期交纳会费的会员”的查询可以用 DATE_ADD()改写为：

```
SELECT last_name, first_name, expiration FROM member
WHERE expiration < DATE_ADD(CURDATE(), INTERVAL 60 DAY);
```

如果为 expiration 列编制索引，将会比先前的查询更有效率，第 5 章将会讨论原因。

在本章前面的内容里，我曾给出一个牙医诊所用来查找“哪些患者没来参加复查”的查询：

```
SELECT last_name, first_name, last_visit FROM patient
WHERE last_visit < DATE_SUB(CURDATE(), INTERVAL 6 MONTH);
```

你当时也许还看不懂这个查询，但现在总该弄明白了吧?

7. 模式匹配

MySQL 支持模式匹配操作，这使我们能够在没有给出精确比较值的情况下把有关的数据行查出来。模式匹配需要使用特殊的操作符（LIKE 和 NOT LIKE），还需要你提供一个包含通配字符的字符串。下划线字符“_”只能匹配一个字符，百分号字符“%”则能匹配任何一个字符序列（包括空序列在内）。

下面这个模式查询将把姓氏以字母 W 或 w 开头的总统姓名查找出来：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE 'W%';
+------------+------------+
| last_name  | first_name |
+------------+------------+
| Washington | George     |
| Wilson     | Woodrow    |
+------------+------------+
```

请大家再来看一个查询，它在使用模式匹配功能时出现了错误；因为它使用的不是 LIKE，而是带有算术比较操作符的模式。

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name = 'W%';
Empty set (0.00 sec)
```

上面这个查询的含义变成了“把姓氏是 W%或 w%的总统找出来”。

下面这个查询将把姓氏里有 W 或 w 字母（并不仅限于姓氏的第一个字母）的总统姓名列出来：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE '%W%';
+------------+------------+
| last_name  | first_name |
+------------+------------+
| Washington | George     |
| Wilson     | Woodrow    |
| Eisenhower | Dwight D.  |
+------------+------------+
```

下面这个查询将把姓氏由且仅由 4 个字母构成的总统姓名给查出来：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name LIKE '____';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Polk      | James K.    |
| Taft      | William H.  |
| Ford      | Gerald R.   |
| Bush      | George H.W. |
| Bush      | George W.   |
+-----------+-------------+
```

MySQL 还提供基于*正则表达式*（regular expression）和 REXEXP 操作符的另一种模式匹配形式，3.5.1 节中的第 1 小节和附录 C 将进一步讨论 LIKE 和 REGEXP 操作符。

8. 如何设置和使用SQL变量

MySQL 允许自定义变量。我们可以使用查询结果来设置变量，这使我们能够方便地把一些值保存起来以供今后查询。例如，你想知道有哪些总统出生在 Andrew Jackson 总统之前。你可以这样做：先检索出他的出生日期并保存到一个变量里，再把出生日期早于这个变量值的总统查找出来：

```
mysql> SELECT @birth := birth FROM president
    -> WHERE last_name = 'Jackson' AND first_name = 'Andrew';
```

```
+------------------+
| @birth := birth |
+------------------+
| 1767-03-15       |
+------------------+
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < @birth ORDER BY birth;
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

变量的命名语法是“@*变量名*”，赋值语法是在 SELECT 语句里使用一个“@*变量名*:= *值*”形式的表达式。因此，上面的第一个查询负责把 Andrew Jackson 总统的出生日期查出来并把它赋值给一个名为@birth 的变量（这条 SELECT 语句的查询结果仍会被显示。把查询结果赋值给一个变量并不会使该查询的输出结果不显示）。第二个查询负责把出生日期早于@birth 变量值的总统查出来。

前面的问题其实可以通过一个联结或子查询在一个查询语句里得到解决，但这里不予讨论。而有时候使用一个变量可能更容易。

SET 语句也能用来对变量赋值，此时，“=”和“:=”都可以用做赋值操作符。如下所示：

```
mysql> SET @today = CURDATE();
mysql> SET @one_week_ago := DATE_SUB(@today, INTERVAL 7 DAY);
mysql> SELECT @today, @one_week_ago;
+------------+---------------+
| @today     | @one_week_ago |
+------------+---------------+
| 2008-03-21 | 2008-03-14    |
+------------+---------------+
```

9. 如何生成统计信息

MySQL 最有用的功能之一是它能够依据大量未经加工的数据生成多种统计汇总信息。大家知道，单纯依靠人工手段来生成统计信息是一项既枯燥又耗时还容易出错的工作。如果大家能掌握使用 MySQL 来生成各种统计信息的技巧，它就会成为你手中最具威力的信息处理工具。

找出一组数据里到底有多少种不同的取值是一项比较常见的统计工作，而关键字 DISTINCT 恰好能让我们把在查询结果中重复出现的数据行清除掉。例如，下面的查询将把美国历届总统的出生地所在州不加重复地列举出来：

```
mysql> SELECT DISTINCT state FROM president ORDER BY state;
+-------+
| state |
+-------+
| AR    |
| CA    |
| CT    |
| GA    |
| IA    |
```

```
| IL    |
| KY    |
| MA    |
| MO    |
| NC    |
| NE    |
| NH    |
| NJ    |
| NY    |
| OH    |
| PA    |
| SC    |
| TX    |
| VA    |
| VT    |
+-------+
```

另一项比较常见的统计工作是利用 COUNT()函数来计数。COUNT(*)能把你的查询到底选取了多少个数据行的情况告诉你。如果你的查询语句没有 WHERE 子句，就会选择所有行 COUNT(*)就会把数据表里的行数告诉你。下面这个查询能让我们知道“美国历史研究会”现在共有多少名会员：

```
mysql> SELECT COUNT(*) FROM member;
+----------+
| COUNT(*) |
+----------+
|      102 |
+----------+
```

如果查询语句带有 WHERE 子句，COUNT(*)会告诉你该子句到底匹配了多少个数据行。例如，下面这个查询能让我们知道你到目前为止已经进行过多少次考试：

```
mysql> SELECT COUNT(*) FROM grade_event WHERE category = 'Q';
+----------+
| COUNT(*) |
+----------+
|        4 |
+----------+
```

COUNT(*)的统计结果是被选中的数据行的总数，而 COUNT(*数据列名称*)值则只统计全体非 NULL 值的个数。这二者之间的区别很容易从下面这个查询看出来：

```
mysql> SELECT COUNT(*), COUNT(email), COUNT(expiration) FROM member;
+----------+--------------+-------------------+
| COUNT(*) | COUNT(email) | COUNT(expiration) |
+----------+--------------+-------------------+
|      102 |           80 |                96 |
+----------+--------------+-------------------+
```

从上面的查询结果可以知道，member 数据表目前共有 102 条记录，其中的 80 条在 email 数据列里有一个值。我们还可以推断出研究会目前有 6 名终身会员（expiration 数据列里的 NULL 值表示这是一名终身会员，因为 102 条记录里有 96 条记录的 expiration 数据列里有非 NULL 值，剩下的 6 条记录就必然属于那些终身会员）。

COUNT()和 DISTINCT 联合起来可以统计出查询结果里到底有多少种不同的非 NULL 值。例如，如

果想知道美国总共有多少个州曾经有总统出生，下面这个查询就能告诉你：

```
mysql> SELECT COUNT(DISTINCT state) FROM president;
+-----------------------+
| COUNT(DISTINCT state) |
+-----------------------+
|                    20 |
+-----------------------+
```

你可以对某个数据列进行全面的统计，也可以对该数据列做分门别类的统计。例如，如果你想知道班级里总共有多少名学生，可以使用下面这个查询：

```
mysql> SELECT COUNT(*) FROM student;
+----------+
| COUNT(*) |
+----------+
|       31 |
+----------+
```

如果想知道班级里分别有多少名男生和女生又该怎么办呢？一种办法是分别对两种性别进行统计，如下所示：

```
mysql> SELECT COUNT(*) FROM student WHERE sex='f';
+----------+
| COUNT(*) |
+----------+
|       15 |
+----------+
mysql> SELECT COUNT(*) FROM student WHERE sex='m';
+----------+
| COUNT(*) |
+----------+
|       16 |
+----------+
```

这个办法管用，但比较麻烦。如果数据列的取值有很多种的话，这个办法就算管用也不适用了。就拿统计出生于美国各个州的总统人数的问题来说吧：你先得一个不落地把有多少个不同的州出生过总统的情况统计出来(`SELECT DISTINCT state FROM president`)，然后再用一系列 `SELECT COUNT(*)` 语句去分别统计出生于各州的总统人数。如此麻烦的事情我想是不会有多少人情愿去做的。

值得庆幸的是，MySQL 可以只用一个查询就把某数据列里的不同值分别出现过多少次的情况统计出来。还是用分别统计男、女学生人数的例子，使用 `GROUP BY` 子句：

```
mysql> SELECT sex, COUNT(*) FROM student GROUP BY sex;
+-----+----------+
| sex | COUNT(*) |
+-----+----------+
| F   |       15 |
| M   |       16 |
+-----+----------+
```

分别统计出生于各州的美国总统人数的问题也可以用类似的办法来解决，如下所示：

```
mysql> SELECT state, COUNT(*) FROM president GROUP BY state;
```

```
+-------+----------+
| state | COUNT(*) |
+-------+----------+
| AR    |        1 |
| CA    |        1 |
| CT    |        1 |
| GA    |        1 |
| IA    |        1 |
| IL    |        1 |
| KY    |        1 |
| MA    |        4 |
| MO    |        1 |
| NC    |        2 |
| NE    |        1 |
| NH    |        1 |
| NJ    |        1 |
| NY    |        4 |
| OH    |        7 |
| PA    |        1 |
| SC    |        1 |
| TX    |        2 |
| VA    |        8 |
| VT    |        2 |
+-------+----------+
```

如果需要进行这种分门别类的统计，GROUP BY 子句就必不可少，它的作用是让 MySQL 知道在统计之前应该如何对有关的数据记录分类。如果你忘了加上这个子句，就会收到一条出错信息。

与反复使用多个彼此近似的查询来分别统计某数据列不同取值出现次数的做法相比，把 COUNT(*) 函数与 GROUP BY 子句相结合的做法有很多优点：

- 在开始统计之前，我们不必知道将被统计的数据列里到底有多少种不同的取值；
- 我们只需使用一个而不是好几个查询；
- 因为只用一个查询就能把所有的结果都查出来，所以我们还能对输出进行排序。

前两项优点的重要性体现在它们有助于简化查询语句的书写。第三项优点的重要性则体现在它能让我们更灵活地显示查询结果。在默认的情况下，MySQL 会根据 GROUP BY 子句里的数据列对查询结果进行排序。但我们也可以通过 ORDER BY 子句按指定顺序排序。例如，如果你想按人数从多到少的顺序把有总统出生的美国各州查出来并排序输出，就可以增加一个如下所示的 ORDER BY 子句：

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state ORDER BY count DESC;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
| NC    |     2 |
| VT    |     2 |
| TX    |     2 |
| SC    |     1 |
```

```
| NH    |       1 |
| PA    |       1 |
| KY    |       1 |
| NJ    |       1 |
| IA    |       1 |
| MO    |       1 |
| CA    |       1 |
| NE    |       1 |
| GA    |       1 |
| IL    |       1 |
| AR    |       1 |
| CT    |       1 |
+-------+-------+
```

当你准备用来排序的输出列是某个汇总函数的计算结果时，不能直接在 ORDER BY 子句里引用函数，而应该先给这个输出列取一个别名，然后再把这个别名用在 ORDER BY 子句里。上面那个查询就是这样做的：我们给 COUNT(*)输出列取了一个别名叫 count。在 ORDER BY 子句里引用输出列的另一种办法是利用它在输出结果中的位置：

```
SELECT state, COUNT(*) FROM president
GROUP BY state ORDER BY 2 DESC;
```

用输出列的出现位置来设定排序顺序的做法在 MySQL 中允许的，但存在着弊病。

- 它很容易导致查询命令难以阅读，因为数字不如文字表意丰富。
- 每当添加、删除、或者调整了输出列的先后次序，都不得不重新检查ORDER BY子句以便对它们的位置编号作相应的更改。
- 在ORDER BY子句中引用列位置的语法，不再属于标准SQL的一部分，是不赞成使用的。

使用别名就不存在这种问题。

类似于 ORDER BY 子句的情况，如果你打算用 GROUP BY 子句对一个计算出来的输出列进行归类，可以使用输出列的别名或者它们在查询结果里的出现位置来设定。下面这个查询把不同月份出生的美国总统的人数分别统计了出来：

这个查询可以用输出列的出现位置改写如下：

```
mysql> SELECT MONTH(birth) AS Month, MONTHNAME(birth) AS Name,
    -> COUNT(*) AS count
    -> FROM president GROUP BY Name ORDER BY Month;
+-------+-----------+-------+
| Month | Name      | count |
+-------+-----------+-------+
|     1 | January   |     4 |
|     2 | February  |     4 |
|     3 | March     |     4 |
|     4 | April     |     4 |
|     5 | May       |     2 |
|     6 | June      |     1 |
|     7 | July      |     4 |
|     8 | August    |     4 |
|     9 | September |     1 |
|    10 | October   |     6 |
|    11 | November  |     5 |
|    12 | December  |     3 |
+-------+-----------+-------+
```

COUNT()函数还能与 ORDER BY 和 LIMIT 子句联合使用，例如，如果你想知道出生总统最多的前 4 个州是哪几个，可以对 president 数据表做如下查询：

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state ORDER BY count DESC LIMIT 4;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
+-------+-------+
```

如果你不想用 LIMIT 子句来限制查询结果中的记录个数，而只是想把与某个特定 COUNT()值相对应的记录找出来，就需要使用一个 HAVING 子句。HAVING 子句与 WHERE 子句的相似之处是它们都是用来设定查询条件的，输出的行必须符合这些查询条件。HAVING 子句与 WHERE 子句的不同之处是 COUNT()之类的汇总函数的计算结果允许在 HAVING 子句里出现。请看下面这个查询，它能告诉我们说美国哪些州有两位或两位以上的总统出生：

```
mysql> SELECT state, COUNT(*) AS count FROM president
    -> GROUP BY state HAVING count > 1 ORDER BY count DESC;
+-------+-------+
| state | count |
+-------+-------+
| VA    |     8 |
| OH    |     7 |
| MA    |     4 |
| NY    |     4 |
| NC    |     2 |
| VT    |     2 |
| TX    |     2 |
+-------+-------+
```

一般说来，带有 HAVING 子句的查询命令特别适合用来查找在某个数据列里重复出现的值（或者不重复出现的值——使用子句 HAVING count = 1 即可）。

除 COUNT()以外，MySQL 还有其他一些汇总函数。函数 MIN()、MAX()、SUM()和 AVG()能够得出某个数据列里的最小值、最大值、总和和平均值。MySQL 允许把它们同时用在同一个查询命令里。下面这个查询对学生们在各次考试或测验中的分数情况作了多种统计处理。结果显示各次考试中有考试成绩的总人数（不包括考试当天缺勤的学生）：

```
mysql> SELECT
    -> event_id,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> MAX(score)-MIN(score)+1 AS span,
    -> SUM(score) AS total,
    -> AVG(score) AS average,
    -> COUNT(score) AS count
    -> FROM score
    -> GROUP BY event_id;
```

```
+----------+---------+---------+------+-------+---------+-------+
| event_id | minimum | maximum | span | total | average | count |
+----------+---------+---------+------+-------+---------+-------+
|        1 |       9 |      20 |   12 |   439 | 15.1379 |    29 |
|        2 |       8 |      19 |   12 |   425 | 14.1667 |    30 |
|        3 |      60 |      97 |   38 |  2425 | 78.2258 |    31 |
|        4 |       7 |      20 |   14 |   379 | 14.0370 |    27 |
|        5 |       8 |      20 |   13 |   383 | 14.1852 |    27 |
|        6 |      62 |     100 |   39 |  2325 | 80.1724 |    29 |
+----------+---------+---------+------+-------+---------+-------+
```

很明显，如果还能知道 event_id 列的值代表的都是考试还是测验，这些信息的意义就更清晰明确了。我们可以做到这一点，但需要查询 grade_event 数据表，我们将在第 10 小节再次讨论这条查询命令。

如果你想输出“统计结果”，那就再增加一条 WITH ROLLUP 子句。这将使 MySQL 对数据行分组统计结果做进一步统计而得到所谓的“超级聚合”值。下面这个简单的例子是早前用来按性别统计学生人数的那条语句的基础上改写而来的，新增加的 WITH ROLLUP 子句将对两种性别的学生人数进行汇总并生成一个输出行：

```
mysql> SELECT sex, COUNT(*) FROM student GROUP BY sex WITH ROLLUP;
+-----+----------+
| sex | COUNT(*) |
+-----+----------+
| F   |       15 |
| M   |       16 |
| NULL|       31 |
+-----+----------+
```

在查询结果里，类别名称列（本例中是 sex 列）里的 NULL 值，表明与它同在一行的数值是它前面那些分组统计结果的汇总统计值。

WITH ROLLUP 子句还可以和其他聚合函数搭配使用。下面这条语句除了像几个段落之前那样对考试成绩进行了几种统计以外，还将生成一个额外的超级统计结果行：

```
mysql> SELECT
    -> event_id,
    -> MIN(score) AS minimum,
    -> MAX(score) AS maximum,
    -> MAX(score)-MIN(score)+1 AS span,
    -> SUM(score) AS total,
    -> AVG(score) AS average,
    -> COUNT(score) AS count
    -> FROM score
    -> GROUP BY event_id
    -> WITH ROLLUP;
+----------+---------+---------+------+-------+---------+-------+
| event_id | minimum | maximum | span | total | average | count |
+----------+---------+---------+------+-------+---------+-------+
|        1 |       9 |      20 |   12 |   439 | 15.1379 |    29 |
|        2 |       8 |      19 |   12 |   425 | 14.1667 |    30 |
|        3 |      60 |      97 |   38 |  2425 | 78.2258 |    31 |
|        4 |       7 |      20 |   14 |   379 | 14.0370 |    27 |
|        5 |       8 |      20 |   13 |   383 | 14.1852 |    27 |
|        6 |      62 |     100 |   39 |  2325 | 80.1724 |    29 |
```

```
|     NULL |       7 |     100 |   94 |  6376 | 36.8555 |   173 |
+----------+---------+---------+------+-------+---------+-------+
```

在上面这份输出结果里，最后一行显示的聚合值是根据它前面的所有分组统计结果值计算出来的。

WITH ROLLUP 子句的有用之处在于它能让你简单方便地获得一些额外的信息，如果不使用它，你恐怕要多进行一次查询才能达到同样的目的。只进行一次查询就能完成任务当然更有效率，因为这可以让 MySQL 服务器不必对数据进行两次甚至更多次的检索。如果 GROUP BY 子句指定了多列，WITH ROLLUP 会生成另外的包含高级统计值的超聚合行。

汇总函数功能很强大，用起来很有趣，但也正因如此，它们也很容易被滥用。请看这个查询：

```
mysql> SELECT
    -> state AS State,
    -> AVG(TIMESTAMPDIFF(YEAR, birth, death)) AS Age
    -> FROM president WHERE death IS NOT NULL
    -> GROUP BY state ORDER BY Age;
+-------+---------+
| State | Age     |
+-------+---------+
| KY    | 56.0000 |
| VT    | 58.5000 |
| NC    | 59.5000 |
| OH    | 62.2857 |
| NH    | 64.0000 |
| NY    | 69.0000 |
| NJ    | 71.0000 |
| TX    | 71.0000 |
| MA    | 72.0000 |
| VA    | 72.3750 |
| PA    | 77.0000 |
| SC    | 78.0000 |
| CA    | 81.0000 |
| MO    | 88.0000 |
| IA    | 90.0000 |
| NE    | 93.0000 |
| IL    | 93.0000 |
+-------+---------+
```

这个查询把所有已经逝世的总统都找了出来，按他们的出生地所在州进行分组，确定每个人逝世时的年龄，然后按各州求出他们逝世时的平均年龄，再按这个平均年龄排序后显示出来。换句话说，这个查询能让我们了解在同一个州出生的总统逝世时的平均年龄。

这又有什么意义呢？它能证明你有写出这类查询命令的能力，但并不能证明这个查询值得你去编写。数据库能让我们做很多事情，但并非每件事情都有意义。当人们发现自己能够利用数据库做很多事情时，他们往往会陷入一种为查询而查询的狂热。这种对统计数字漫无目标的热衷在最近几年的体育赛事电视转播中表示得尤其明显。利用他们的数据库，赛事统计人员能告诉你很多关于比赛你确实想知道的事情，但同时也能告诉你很多你根本就不想知道或者根本就没有想到过的事情。例如，你真的关心橄榄球队里哪个四分卫替补球员在他所属的球队领先对手 14 多分的情况下，在赛事第二节的最后两分钟里，在 15 码线区域内阻截对手达阵的次数最多吗？

10. 如何从多个数据表里检索信息

到目前为止，我们查询出来的信息都来自一个数据表。MySQL 的能力其实远不止此。前面说过，RDBMS（关系数据库管理系统）的威力在于它们能把一种东西与另一种东西关联起来，即能把来自多个数据表的信息结合在一起以解答单个数据表不足以解答的问题。本节将介绍涉及多个数据表的查询命令的编写方法。

当你打算从多个数据表选取信息时，有一种方法叫做*联结*（join）。这种叫法是因为必须把一个数据表与另一个数据表中的信息结合起来才能得到查询结果。联结操作是通过把两个（或多个）数据表里的同类数据进行匹配而完成的。多表操作的另一种方法就是将 SELECT 语句嵌套在另一个 SELECT 语句里，前者叫做*子查询*。本节将介绍这两种操作。

我们先来看一个关于联结的例子。在 1.4.6 节中第 2 小节里，我给出了一个用来检索给定日期考试或测验分数的查询命令，但我当时没有对它进行解释。现在是解释那条查询命令的时候了。那条查询命令实际上涉及 3 个数据表，即需要进行一次三方联结操作。现在，我们分两步来对它进行说明。首先，我们来构造一个能查出给定日期的分数的查询命令：

```
mysql> SELECT student_id, date, score, category
    -> FROM grade_event INNER JOIN score
    -> ON grade_event.event_id = score.event_id
    -> WHERE date = '2008-09-23';
+------------+------------+-------+----------+
| student_id | date       | score | category |
+------------+------------+-------+----------+
|          1 | 2008-09-23 |    15 | Q        |
|          2 | 2008-09-23 |    12 | Q        |
|          3 | 2008-09-23 |    11 | Q        |
|          5 | 2008-09-23 |    13 | Q        |
|          6 | 2008-09-23 |    18 | Q        |
...
```

这个查询先查出给定日期('2008-09-23')的 grade_event 行，再利用此行里的 event_id（考试事件编号）把 score 数据表里拥有同一 event_id 的考试分数都查出来。每找到一组彼此匹配的 grade_event 行和 score 行，就把学生的学号、考试分数、日期和考试事件的类型显示出来。

与以前介绍的查询命令相比，这个查询在以下几方面有着显著的区别。

- 在FROM子句里，我们列举了多个数据表的名字，因为我们要从多个数据表里检索信息：

  ```
  FROM grade_event INNER JOIN score
  ```

- 在ON子句里，我们给出了grade_event数据表和score数据表的联结条件：这两个数据表里的event_id列的值必须相互匹配：

  ```
  ON grade_event.event_id = score.event_id
  ```

请注意我在将 event_id 数据列命名为 grade_event、event_id 和 score.event_id 时所使用的 *tbl_name.col_name* 语法，这是为了让 MySQL 知道我们提到的数据表到底是哪几个。（因为两个数据表都有 event_id 数据列，所以如果不加上数据表的名字以区分，就会产生二义性。）但这个查询命令中的其他数据列（即 date、score、type）却用不着加上数据表的名字以进行区分，因为它们在不同的数据表里只出现一次，不可能产生二义性。

我个人喜欢在每个数据列的前面都加上数据表的名字以示区分。在今后涉及联结操作的查询示例

里，我将一直沿用这个习惯。在给每一个数据列都加上它们各自所属的数据表名字之后，这个查询将成为如下所示的样子：

```
SELECT score.student_id, grade_event.date, score.score, grade_event.category
FROM grade_event INNER JOIN score
ON grade_event.event_id = score.event_id
WHERE grade_event.date = '2008-09-23';
```

第一阶段的查询利用 grade_event 数据表把日期映射到一个考试事件编号，再利用这个考试事件编号把 score 数据表里与自己相匹配的考试分数找出来。这个查询的输出结果只能让我们看到 student_id 数据列的值，要是能把学生们的姓名直接列出来就更清晰易懂了。第二阶段，利用 student 数据表把学生们的学号映射为他们的姓名。score 和 student 数据表都有 student_id 数据列，两个数据表里的记录能够通过这个数据列被关联起来，利用这一事实就能把学生们的姓名也显示出来。如下所示：

```
mysql> SELECT
    -> student.name, grade_event.date, score.score, grade_event.category
    -> FROM grade_event INNER JOIN score INNER JOIN student
    -> ON grade_event.event_id = score.event_id
    -> AND score.student_id = student.student_id
    -> WHERE grade_event.date = '2008-09-23';
+-----------+------------+-------+----------+
| name      | date       | score | category |
+-----------+------------+-------+----------+
| Megan     | 2008-09-23 |    15 | Q        |
| Joseph    | 2008-09-23 |    12 | Q        |
| Kyle      | 2008-09-23 |    11 | Q        |
| Abby      | 2008-09-23 |    13 | Q        |
| Nathan    | 2008-09-23 |    18 | Q        |
...
```

与以前介绍的查询命令相比，这个查询在以下几方面有着显著的区别。

- 在FROM子句里，除grade_event和score数据表外，我们又增加了student数据表。
- 在前一个查询里，student_id数据列不会产生二义性，所以我们当时既可以不给它加上数据表的名字（即写成student_id的形式），也可以给它加上数据表的名字（即写成score.student_id的形式）。但在这个查询里，因为score和student数据表都有student_id数据列，为了避免产生二义性，必须把它分别写成score.student_id和student.student_id以表明它们来自不同的数据表。
- ON子句里多了一个查询条件：score数据表里的行必须在student_id数据列上与student数据表里的行相匹配。

  ```
  ON ... score.student_id = student.student_id
  ```

- 查询结果将列出学生们的姓名而不是学号。（当然，如果你愿意，也可以把它们同时显示出来，只需要在输出列的列表加上student.student_id。）

只要对这个查询里的日期值进行替换，就能查出任何一天的考试分数、参加考试的学生名单以及考试的类型。你根本用不着知道学生学号或考试事件编号之类的东西，因为 MySQL 将自动查出有关的 ID 值并利用它们把你想要的信息找出来。

考试记分项目中的另一项工作是统计学生们的考试缺勤情况，并以学生学号和考试日期的形式记

录在 absence 数据表里。如果你想看到缺勤学生的姓名（而不仅仅是学号），就需要把 absence 表与 student 表通过 student_id 列的值联结起来。下面这个查询将列出缺勤学生的学号、姓名和他们的缺勤次数。

```
mysql> SELECT student.student_id, student.name,
    -> COUNT(absence.date) AS absences
    -> FROM student INNER JOIN absence
    -> ON student.student_id = absence.student_id
    -> GROUP BY student.student_id;
+------------+-------+----------+
| student_id | name  | absences |
+------------+-------+----------+
|          3 | Kyle  |        1 |
|          5 | Abby  |        1 |
|         10 | Peter |        2 |
|         17 | Will  |        1 |
|         20 | Avery |        1 |
+------------+-------+----------+
```

注意 在这个查询命令的 GROUP BY 子句里，数据列名字前面也加上了数据表的名字，但就这个查询而言，这并不是必要的。这是因为：GROUP BY 子句作用于输出列，而这里的查询结果只包含一个名为 student_id 的列，所以 MySQL 知道你指的是哪一个。

如果只想知道有哪些学生缺席，这个查询的结果将正是我们需要的。但如果你还要把这份名单交到学校办公室去，他们可能会问："其他学生的出勤情况呢？我们想知道每一名学生的出勤情况。"这是一个稍微不同的问题，他们想知道每一名学生（包括参加了考试的学生）的缺勤次数。但因为问法不同，回答这个问题使用的查询也就不同。

为了回答这个问题，我们可以用 LEFT JOIN 来代替普通的联结操作。LEFT JOIN 将使 MySQL 为联结操作中第一个数据表（即名称出现在关键字 LEFT JOIN 左边的那个数据表）里的每一个中选数据行生成一个输出行。只要先列出 student 数据表，我们就能得到全体学生（包括那些没有在 absence 表里出现过的学生）的考试出勤情况。这个查询的具体写法是这样的：在 FROM 子句里用关键字 LEFT JOIN（而不是逗号）来分隔各数据表的名字，再增加一个 ON 子句来指定两个数据表中的数据记录的匹配关系。如下所示：

```
mysql> SELECT student.student_id, student.name,
    -> COUNT(absence.date) AS absences
    -> FROM student LEFT JOIN absence
    -> ON student.student_id = absence.student_id
    -> GROUP BY student.student_id;
+------------+-----------+----------+
| student_id | name      | absences |
+------------+-----------+----------+
|          1 | Megan     |        0 |
|          2 | Joseph    |        0 |
|          3 | Kyle      |        1 |
|          4 | Katie     |        0 |
|          5 | Abby      |        1 |
|          6 | Nathan    |        0 |
```

```
|          7 | Liesl     |         0 |
...
```

在前面的第 9 小节里，曾有一个对 score 数据表里的数据进行各种统计分析的查询。那个查询的输出结果列出了考试事件的编号，但因为我们当时还不知道如何联结 score 数据表与 grade_event 数据表才能把考试事件编号映射为考试的日期和类型，所以那份查询结果就没有包括考试的日期和类型。现在，我们知道应该怎样做了。下面这个查询与前面那个差不多，但当初那个简单的考试事件编号数字现在已经被取代为相应的日期和类型了：

```
mysql> SELECT
    -> grade_event.date,grade_event.category,
    -> MIN(score.score) AS minimum,
    -> MAX(score.score) AS maximum,
    -> MAX(score.score)-MIN(score.score)+1 AS span,
    -> SUM(score.score) AS total,
    -> AVG(score.score) AS average,
    -> COUNT(score.score) AS count
    -> FROM score INNER JOIN grade_event
    -> ON score.event_id = grade_event.event_id
    -> GROUP BY grade_event.date;
+------------+----------+---------+---------+------+-------+---------+-------+
| date       | category | minimum | maximum | span | total | average | count |
+------------+----------+---------+---------+------+-------+---------+-------+
| 2008-09-03 | Q        |       9 |      20 |   12 |   439 | 15.1379 |    29 |
| 2008-09-06 | Q        |       8 |      19 |   12 |   425 | 14.1667 |    30 |
| 2008-09-09 | T        |      60 |      97 |   38 |  2425 | 78.2258 |    31 |
| 2008-09-16 | Q        |       7 |      20 |   14 |   379 | 14.0370 |    27 |
| 2008-09-23 | Q        |       8 |      20 |   13 |   383 | 14.1852 |    27 |
| 2008-10-01 | T        |      62 |     100 |   39 |  2325 | 80.1724 |    29 |
+------------+----------+---------+---------+------+-------+---------+-------+
```

对涉及多个数据表的查询结果里的输出列也可以使用 COUNT() 和 AVG() 等汇总函数。请看下面这个查询，它将把与考试日期与考生性别的每一种组合相对应的考生人数（即考试分数的个数）和平均分数统计出来：

```
mysql> SELECT grade_event.date, student.sex,
    -> COUNT(score.score) AS count, AVG(score.score) AS average
    -> FROM grade_event INNER JOIN score INNER JOIN student
    -> ON grade_event.event_id = score.event_id
    -> AND score.student_id = student.student_id
    -> GROUP BY grade_event.date, student.sex;
+------------+-----+-------+---------+
| date       | sex | count | average |
+------------+-----+-------+---------+
| 2008-09-03 | F   |    14 | 14.6429 |
| 2008-09-03 | M   |    15 | 15.6000 |
| 2008-09-06 | F   |    14 | 14.7143 |
| 2008-09-06 | M   |    16 | 13.6875 |
| 2008-09-09 | F   |    15 | 77.4000 |
| 2008-09-09 | M   |    16 | 79.0000 |
| 2008-09-16 | F   |    13 | 15.3077 |
| 2008-09-16 | M   |    14 | 12.8571 |
| 2008-09-23 | F   |    12 | 14.0833 |
```

```
| 2008-09-23 | M   |     15 | 14.2667 |
| 2008-10-01 | F   |     14 | 77.7857 |
| 2008-10-01 | M   |     15 | 82.4000 |
+------------+-----+-------+---------+
```

考试记分项目的另一项工作是计算每位学生的期末总成绩，这也可以用一个类似的查询来完成。下面就是完成这一工作的查询命令：

```
SELECT student.student_id, student.name,
SUM(score.score) AS total, COUNT(score.score) AS n
FROM grade_event INNER JOIN score INNER JOIN student
ON grade_event.event_id = score.event_id
AND score.student_id = student.student_id
GROUP BY score.student_id
ORDER BY total;
```

联结操作并非只能作用于不同的数据表，这乍听起来有点奇怪，但你确实能把某个数据表与它自身结合起来。例如，如果你想知道是否有两位（或者多位）总统出生于同一城市，就需要检查每位总统的出生地是否与其他总统的出生地一样。下面就是完成这一查询的命令：

```
mysql> SELECT p1.last_name, p1.first_name, p1.city, p1.state
    -> FROM president AS p1 INNER JOIN president AS p2
    -> ON p1.city = p2.city AND p1.state = p2.state
    -> WHERE (p1.last_name <> p2.last_name OR p1.first_name <> p2.first_name)
    -> ORDER BY state, city, last_name;
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Adams     | John Quincy | Braintree | MA    |
| Adams     | John        | Braintree | MA    |
+-----------+-------------+-----------+-------+
```

这个查询命令有两个地方特别值得注意，如下所示。

- 它需要两次用到同一个数据表，所以我们必须为它创建两个别名（p1和p2）才能把表中的同名数据列区别开来。有了列别名，在为表另定义别名时，AS关键字就是可选的。
- 每位总统的记录都将与其本身相匹配，但这并不是我们想要的输出结果。WHERE子句通过确保每位总统的记录只能与其他总统的记录比较，来剔除“记录与它本身相匹配”的情况。

用一个类似的查询可以查出在同一天出生的总统。可是，如果直接比较某两位总统的出生日期，就只能把同年同月同日出生的总统查出来，那些生日在同一天但出生年份不同的总统将不会出现在查询结果里。因此，我们必须用 MONTH()和 DAYOFONTH()函数比较出生日期值中的月份和日期，如下所示：

```
mysql> SELECT p1.last_name, p1.first_name, p1.birth
    -> FROM president AS p1 INNER JOIN president AS p2
    -> WHERE MONTH(p1.birth) = MONTH(p2.birth)
    -> AND DAYOFMONTH(p1.birth) = DAYOFMONTH(p2.birth)
    -> AND (p1.last_name <> p2.last_name OR p1.first_name <> p2.first_name)
    -> ORDER BY p1.last_name;
+-----------+------------+------------+
| last_name | first_name | birth      |
+-----------+------------+------------+
| Harding   | Warren G.  | 1865-11-02 |
```

```
| Polk       | James K.   | 1795-11-02 |
+------------+------------+------------+
```

用 DAYOFYEAR()来代替 MONTH()与 DAYOFONTH()的组合会使我们得到一个稍微简单点儿的查询命令，但查询结果却可能是不正确的，因为没有考虑到闰年的因素。

进行多表检索的另一类办法是使用“子查询”，也就是把一条 SELECT 语句嵌套在另一条里。子查询有好几种类型，我们将在 2.9 节中进一步讨论。就目前而言，只要熟悉几个例子就可以了。我们不妨假设需要把出全勤的学生都找出来。这等价于把没在 absence 数据表里出现过的学生找出来，我们可以用如下所示的语句来达到目的：

```
mysql> SELECT * FROM student
    -> WHERE student_id NOT IN (SELECT student_id FROM absence);
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Katie     | F   |          4 |
| Nathan    | M   |          6 |
| Liesl     | F   |          7 |
...
```

嵌套于内层的 SELECT 语句用来生成一个在 absence 数据表里出现过的 student_id 值的集合，外层的 SELECT 语句负责把与该集合里的任何一个 ID 值都不匹配的 student 数据行检索出来。

利用子查询可以为 1.4.9 节的第 8 小节里提出的一个问题提供一个单语句解决方案，那个问题是哪些总统出生在 Andrew Jackson 之前。当初的解决方案使用了两条语句和一个用户变量，而我们现在可以用一个如下所示的子查询来解决它：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth < (SELECT birth FROM president
    -> WHERE last_name = 'Jackson' AND first_name = 'Andrew');
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
| Adams      | John       | 1735-10-30 |
| Jefferson  | Thomas     | 1743-04-13 |
| Madison    | James      | 1751-03-16 |
| Monroe     | James      | 1758-04-28 |
+------------+------------+------------+
```

内层 SELECT 语句用来确定 Andrew Jackson 的出生日期，外层 SELECT 语句负责检索生日早于该日期的总统。

1.4.10　如何删除或更新现有的数据行

有时候，你需要删除或修改一些现有的数据行，这就需要用到 DELETE 和 UPDATE 语句。本节的讨论重点就是这两条语句的用法。

DELETE 语句的基本格式如下所示：

```
DELETE FROM tbl_name
WHERE which rows to delete;
```

WHERE 子句是可选的，它指定哪些数据行会被删除掉；而如果没有 WHERE 子句，数据表里的全部行将都被删除掉。这意味着形式越简单的 DELETE 语句往往越危险，例如：

```
DELETE FROM tbl_name;
```

这条语句将把数据表里的内容删除得一干二净。请千万小心！如果只想删除满足部分数据记录，就必须用 WHERE 子句把它们先筛选出来。这与 SELECT 语句里用 WHERE 子句来避免选取整个数据表的做法类似。例如，如果想删除美国总统的记录，就应该使用下面这样的查询：

```
mysql> DELETE FROM president WHERE state='OH';
Query OK, 7 rows affected (0.00 sec)
```

如果不清楚某条 DELETE 语句到底会删掉哪些数据行，最好先把这条语句的 WHERE 子句放到一条 SELECT 语句里，看这条 SELECT 语句能查出哪些记录。这让你可以确认那些将被删除的行都是你想要删除的，既不多也不少。例如，如果想把 Teddy Roosevelt 总统的记录删掉，能不能使用下面这个查询呢？

```
DELETE FROM president WHRER last_name='Roosevelt';
```

这个语句确实会把 Teddy Roosevelt 总统的记录删掉，但它同时还会把 Franklin Roosevelt 总统的记录也删掉。因此，为保险起见，最好先把这个 WHERE 子句放到 SELECT 语句里检查一下，如下所示：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='Roosevelt';
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Roosevelt | Theodore    |
| Roosevelt | Franklin D. |
+-----------+-------------+
```

从上面的查询结果可以看出，还需要对总统的名字做更细致的设定：

```
mysql> SELECT last_name, first_name FROM president
    -> WHERE last_name='Roosevelt' AND first_name='Theodore';
+-----------+------------+
| last_name | first_name |
+-----------+------------+
| Roosevelt | Theodore   |
+-----------+------------+
```

现在，你应该知道要用什么样的 WHERE 子句才能选出打算删除的行了。下面是改正后的 DELETE 语句：

```
mysql> DELETE FROM president
    -> WHERE last_name='Roosevelt' AND first_name='Theodore';
```

不过，要是每删除一行都得这么办，那可就太麻烦了。但麻烦总比后悔好，对吧？如果遇到这类场合，可以通过复制加粘贴或者输入行编辑功能来尽可能地减少重复打字动作。1.5 节将对这方面的技巧进行介绍。

如果想修改现有行，就需要使用 UPDATE 语句，它的基本格式如下：

```
UPDATE tbl_name
```

```
SET which columns to change
WHERE which rows to update;
```

类似于 DELETE 语句中的情况，这里的 WHERE 子句也是可选的。如果没有给出 WHERE 子句，就表示该数据表里的每一条记录都需要修改。例如，下面这个查询将把每个学生的名字都改成 Georage：

```
mysql> UPDATE student SET name='George';
```

显然，必须谨慎对待这类查询，所以通常都加上一条 WHERE 子句来限制哪些行需要修改。我们来看一个“美国历史研究会”场景中的例子：研究会新吸收了一名会员，但在添加他的个人资料记录项时，只填写了几个数据列。

```
mysql> INSERT INTO member (last_name,first_name)
    -> VALUES('York','Jerome');
```

你发现自己忘了给他设置会员资格失效日期。可以用一条 UPDATE 语句来弥补，其中包含合适的 WHERE 子句来找到想要修改的行：

```
mysql> UPDATE member
    -> SET expiration='2009-7-20'
    -> WHERE last_name='York' AND first_name='Jerome';
```

可以用一条语句修改多个数据列。下面这条 UPDATE 语句将修改 Jerome 的电子邮件地址和普通邮政地址：

```
mysql> UPDATE member
    -> SET email='jeromey@aol.com', street='123 Elm St',
    -> city='Anytown', state='NY', zip='01003'
    -> WHERE last_name='York' AND first_name='Jerome';
```

如果某个数据列允许使用 NULL 值，可以把它设置为 NULL，从而使它处于“未设置”状态。例如，假如 Jerome 在日后一次性地交齐了足以让他成为终身会员的会费，就应该把他的会员资格失效日期设置为 NULL（意思是永不失效）：

```
mysql> UPDATE member
    -> SET expiration=NULL
    -> WHERE last_name='York' AND first_name='Jerome';
```

类似于 DELETE 语句，为了确保能不多不少地把你想要修改的记录全都筛选出来，最好先把 UPDATE 语句的 WHERE 子句放到一条 SELECT 语句里去检查。如果检索条件偏于严格或宽松，就会出现少修改或多修改了一些数据记录的情况。

如果你练习了本节里的查询命令，你的 sampdb 数据库里有关的数据表就会有一些记录删除或者修改掉了。在开始学习下一节之前，应该把那些改动都恢复到原状。如果真的需要重新加载那些数据表，请参考 1.4.8 节中的说明。

1.5　与客户程序 mysql 交互的技巧

本节将介绍一些与客户程序 mysql 交互的技巧，这些技巧能帮助我们更有效率地完成任务，让我们少打一些字。此外，还将学习到怎样更加方便迅速地连接服务器，怎样才能避免逐条输入查询命令，等等。

1.5.1　简化连接过程

在启动 mysql 程序时，通常都需要设定一些连接参数，如主机名、用户名或口令等。如果在每次启动 mysql 程序时都输入这么多的内容，你很快就会感到厌烦，也很容易打错字。其实，在连接 MySQL 服务器时，有好几种办法能减少必需的打字输入内容：

- 把连接参数保存在一个选项文件里；
- 利用shell的命令历史功能重复输入命令；
- 利用shell别名或脚本为mysql命令行定义一个快捷方式。

1. 使用一个选项文件

MySQL 允许把连接参数保存到一个选项文件（option file）里。这样，就用不着在每次启动 mysql 程序时都亲自输入这些参数了。系统将从选项文件里读入有关的参数，就好像已经在命令行上输入了它们一样。这样做的好处是其他 MySQL 客户程序（如 mysqlimport 或 mysqlshow）也能使用这些参数。换句话说，选项文件不仅能简化 mysql 程序的启动过程，也使很多其他程序的启动过程变得简单了。本节简要介绍如何设置选项文件以供客户程序使用，其他内容请参见 F.2.2 节。

在 Unix 系统上，你可以创建一个名为~/.my.cnf 的文件（即在登录主目录里创建一个名为.my.cnf 的文件）来作为你的选项文件。在 Windows 系统上，这个选项文件可以被创建为 MySQL 安装目录下的 my.ini 文件或者 C 盘根目录下的 C:\my.cnf 文件。这个选项文件是一个纯文本文件，所以可以用任何一种文本编辑器来创建。这个文件的内容应该是下面这个样子：

```
[client]
host=server_host
user=your_name
password=your_pass
```

其中，[client]是 MySQL 客户程序选项组的开始标记，由此往后直到文件结尾或下一个选项组开始标记的那些文本行，将逐一给出 MySQL 客户程序在启动时需要用到的选项值。请把其中的 server_host、your_name 和 your_pass 分别替换为你在连接 MySQL 服务器时使用的主机名、用户名和口令。例如，如果服务器在主机 cobra.snake.net 上运行，MySQL 用户名和口令分别是 sampadm 和 secret，下面就是.my.cnf 文件的内容：

```
[client]
host=cobra.snake.net
user=sampadm
password=secret
```

[client]是 MySQL 客户程序选项组的开始标记，它不能省略。但那些用来定义连接参数值的文本行却都是可选的，你可以只列举你需要的连接参数。例如，假如你使用的是 Unix 系统，而你的 MySQL 用户名又与你的 Unix 登录名一样，就不需要包括 user 那一行。默认主机是本地主机（localhost），你只打算连接到本地主机上的服务器，就不需要 host 那一行。

如果是在 Unix 系统上，那么在创建选项文件之后，还需要再多做一项工作：给这个选项文件设置访问权限，以保证别人不会读取或者修改它。下面两条命令都可以把这个文件的访问权限设置为只允许你本人访问：

```
% chmod 600 .my.cnf
% chmod u=rw,go-rwx .my.cnf
```

2. 利用shell的命令历史功能

有些 shell（如 tcsh 或 bash 等）能把你在命令行上输入过的命令记在一个命令历史清单里，你能查看并反复多次地选用其中的命令。如果 shell 具备这一功能，你就可以利用这份命令历史清单来避免敲入大段的命令内容。例如，如果最近使用过 mysql 客户程序，就可以像下面这样把它从命令历史清单里找出来并重新执行一遍：

```
% !my
```

感叹号字符的作用是：让 shell 从命令历史清单里把你最近输入过的、以 my 开头的命令找出来并重新执行一遍，就好像你在命令行上再次输入了这条命令一样。有些 shell 还可以用于利用键盘的上下箭头键（或 Ctrl-P、Ctrl-N 组合键）在命令历史清单里前后移动，当找到你想要执行的命令后，按下 Enter 键即可执行它。tcsh 和 bash 有这种功能，其他 shell 可能也有。你的 shell 是否具备命令历史功能以及该功能的具体使用方法可以在 shell 的帮助文档里查到。

3. 利用shell别名和脚本

如果 shell 允许使用别名机制，你就能把一个较短的命令映射为一条较长的命令。例如，如果使用的 shell 是 csh 或 tcsh，你就可以像下面这样用 alias 命令来定义出一个“新”的 sampdb 命令来，如下所示：

```
alias sampdb 'mysql -h cobra.snake.net -p -u sampadm sampdb'
```

如果使用的 shell 是 bash，定义语法将稍有不同：

```
alias sampdb='mysql -h cobra.snake.net -p -u sampadm sampdb'
```

完成别名定义工作之后，下面两条命令将完全等价：

```
% sampdb
% mysql -h cobra.snake.net -p -u sampadm sampdb
```

很明显，第一个命令比第二个简短得多。如果想让这个别名在你每次登录系统时都能生效，就需要把 alias 命令放到 shell 的某个启动文件（如 tcsh 下的.tcshrc 文件，或者 bash 下的.bashrc 或.bash_profile 文件）里。

Windows 系统上也有类似的技巧：先为 mysql 程序创建一个快捷方式，再通过编辑该快捷方式属性把相关的连接参数包括进去。

还有一个办法能让你在调用命令时少打些字：创建一个脚本，用合适的选项执行 mysql。下面就是一个与刚才定义的命令别名 sampdb 等价的 shell 脚本（适用于 Unix 系统）：

```
#!/bin/sh
exec mysql -h cobra.snake.net -p -u sampadm sampdb
```

如果把这个脚本命名为 sampdb 并（用 chmod +x sampdb 命令）设置为可执行，那么在命令提示符处敲入 sampdb 就能启动 mysql 程序并连接到 sampdb 数据库。

在 Windows 系统上可以利用批处理文件做同样的事情。创建一个名为 sampdb.bat 的批处理文件，再把下面这行文字输入其中：

```
mysql -h cobra.snake.net -p -u sampadm sampdb
```

此后，执行这个批处理文件的办法有两种：一是在控制台窗口敲入 sampdb，二是双击这个批处理文件的 Windows 图标。

如果需要访问多个数据库或连接到多个主机，不妨多定义几个别名、快捷方式或者脚本，让它们以不同的选项参数来启动 mysql 程序。

1.5.2 减少输入查询命令时的打字动作

从对数据库进行交互式查询的角度讲，mysql 是一个非常有用的程序，但它的操作界面却最适合用来输入短小的单行查询命令。虽说 mysql 本身并不关心我们输入的查询命令是否会延续好几行，但输入一条长长的查询命令却不是件会让人高兴的事。如果因为语法错误而不得不重新输入一遍，你应该会很郁闷。有好几种办法能帮助我们减少不必要的打字录入工作：

- ❑ 利用mysql的输入行编辑功能；
- ❑ 利用“复制+粘贴”来发出查询命令；
- ❑ 以批处理模式运行mysql程序。

1. 利用mysql的输入行编辑器

mysql 程序内建有 GNU Readline 库的输入行编辑功能。你可以编辑当前输入行的内容，也可以把以前的输入行调出来直接或经修改之后再次输入。当你在自己输入的命令行里发现了打字错误并想及时纠正时，这非常适用。在按 Enter 键之前，你可以把光标移到出错位置并改正那个打字错误。如果你在按下 Enter 键之后才发现有打字错误，可以把它调出来并在改正错误之后再次提交。（如果查询命令只有一行，改起来就更容易了。）

表 1-4 列出了一些有用的输入行编辑功能的按键序列，除了这些，还有很多常见的输入行编辑命令。你可以在 bash 使用手册介绍命令行编辑功能的有关章节里查到一份完整的清单，在线版 bash 使用手册可以在 GNU 项目的网站 http://www.gnu.org /manual/上找到。

表1-4 mysql程序的输入编辑命令

按键序列	含 义
上箭头键 或 Ctrl-P	调出前一个输入行
下箭头键 或 Ctrl-N	调出后一个输入行
左箭头键 或 Ctrl-B	向左移动光标
右箭头键 或 Ctrl-F	向右移动光标
Escape b	把光标向后移动一个单词
Escape f	把光标向前移动一个单词
Ctrl-A	把光标移到输入行的开头
Ctrl-E	把光标移到输入行的末尾
Ctrl-D	删除光标位置上的那个字符
Delete	删除光标前面（左侧）的那个字符
Escape D	删除单词
Escape Backspace	删除光标前面（左侧）的那个单词
Ctrl-K	删除从光标位置到输入行末尾的所有内容
Ctrl-_	取消前一次修改（可多次重复）

Readline 库没有适用于 Windows 平台的版本，所以在 Windows 平台上无法使用 Readline 库提供的编辑功能。幸好 Windows 本身支持表 1-5 里的编辑命令，因而在 mysql 工具的命令行上也可以使用它们。

表1-5　Windows平台上的输入编辑命令

按键序列	含　义
上箭头	调出前一行
下箭头	调出后一行
左箭头	光标左移一个字符（后退）
右箭头	光标右移一个字符（前进）
Ctrl + 左箭头	光标左移一个单词
Ctrl + 右箭头	光标右移一个单词
Home	光标移动到行首
End	光标移动到行尾
Delete	删除光标处的字符
Backspace（退格键）	删除光标左边的字符
Esc	删除整行
Page Up	调出最早输入的命令
Page Down	调出最后输入的命令
F3	调出最后输入的命令
F7	弹出命令菜单，用上箭头/下箭头键选择
F9	弹出命令菜单，用命令编号选择
F8，F5	循环显示命令列表

下面是输入行编辑功能一个简单的用法示例。假设你在 mysql 程序里输入了如下所示的查询命令：

```
mysql> SHOW COLUMNS FROM persident;
```

在按 Enter 键之前，如果你注意到自己把 president 错误地输成 persident 了，可以像下面这样修改查询。先按几次左箭头键把光标左移到字符 s 的位置上。按两次 Delete 或 Backspace 键，这两个键都可以删除你的系统上光标左侧的字符以删除 er，再重新输入 re 以改正错误。然后按下 Enter 键以提交修改后的查询命令。如果你没有在按下 Enter 键之前发现这个打字错误也不要紧。等看到 mysql 显示的出错信息后，按上箭头键调出刚刚输入的查询命令，然后再按上述过程修改就可以了。

2. 利用“复制+粘贴”来查询

如果你在一个窗口化的操作环境里工作，可以通过“复制+粘贴”操作把你认为有价值的查询命令保存到一个文件里供今后使用。整个操作步骤如下所示。

(1) 在一个终端窗口启动 mysql 程序。

(2) 在一个文档窗口里打开用来存放查询命令的文件（例如，在 Unix 系统上，我将使用 vi。在 Windows 系统上，我将使用 gvim）。

(3) 在文件里找到你想要执行的查询命令，选取并复制下来。再切换到终端窗口，把刚才复制下来的查询命令粘贴到 mysql 程序的提示符处。

这一过程看起来比较繁琐，但熟练掌握之后却相当快捷。它能让你不需打字，迅速地输入一条查询命令。

还可以把“复制+粘贴”操作反过来用（即把有关命令从终端窗口复制到你的查询命令存档文件里）。在 Unix 系统，当你在 mysql 程序里输入查询命令时，它们会被保存到你登录主目录里的一个名为.mysql_history 的文件里。如果你想把自己输入的某个查询命令保存起来供今后使用，可以这样

做：退出 mysql 程序，用一个文本编辑器打开.mysql_history 文件，找到这条查询命令并把它从.mysql_history 文件“复制+粘贴”到查询命令存档文件去。

3. 用mysql程序执行脚本文件

mysql 程序并非只能运行在交互模式下。mysql 程序能够以非交互（即批处理）模式运行并从文件里读取输入。如果你有一些需要定期运行的查询命令，这个技巧将特别有用，你再也用不着每次运行时都重新敲入它们了。只要把命令在文件中保存一次，就可以反复多次地让 mysql 程序根据需要执行它们了。

来看一个“美国历史研究会”场景中的查询示例。假设需要通过 member 数据表里的 interests 数据列来查找哪些会员对美国历史上的某个特定事件感兴趣。例如，为了了解哪些会员对 Great Depression（美国在 1930 年代的大萧条时期）感兴趣，可以编写下面这样的查询命令：

```
SELECT last_name, first_name, email, interests FROM member
WHERE interests LIKE '%depression%'
ORDER BY last_name, first_name;
```

你把这个查询命令保存在 interests.sql 文件里，将文件馈入 mysql 程序里就可以运行它了：

```
% mysql  sampdb < interests.sql
```

在默认的情况下，以批处理模式运行的 mysql 程序的输出内容是以制表符来分隔的。如果想得到与你以交互方式运行 mysql 程序时的输出格式相同的整齐效果，就需要增加一个-t 选项，如下所示：

```
% mysql -t sampdb < interests.sql
```

如果想把输出结果保存起来，可以把它重定向到一个文件里，如下所示：

```
% mysql -t sampdb < interests.sql > interests.out
```

如果你已经在运行 mysql 了，可通过 source 命令执行文件内容：

```
mysql> source interests.sql
```

如果需要了解哪些会员对 Thomas Jefferson 总统的生平感兴趣，只需把 depression 改为 Jefferson 并再次运行 mysql 程序即可。不过，这个办法只有在你不需要非常频繁地查询时才显得有优势。如果必须频繁地运行某个查询命令，还需要找一个更好的办法。在 Unix 上，增加查询命令的灵活性的办法之一是把它保存为一个能够接受命令行参数的脚本，这个脚本将根据你给出的命令行参数对查询命令的具体内容作出修改，进而完成不同的查询任务。这样可以为查询确定参数，在运行脚本时你就可以指定 interests 值。以下面的 shell 脚本 interests.sh 为例：

```
#!/bin/sh
# interests.sh - find USHL members with particular interests
if [ $# -ne 1 ]; then echo 'Please specify one keyword'; exit; fi
mysql -t sampdb <<QUERY_INPUT
SELECT last_name, first_name, email, interests FROM member
WHERE interests LIKE '%$1%'
ORDER BY last_name, first_name;
QUERY_INPUT
```

这个脚本程序的第 3 行确保命令行参数只有一个，否则，它就会打印一条简短的出错信息并退出执行。<<QUERY_INPUT 到脚本程序结尾处的 QUERY_INPUT 之间的文字将成为 mysql 程序的输入。shell 会把这段查询命令文本里的脚本变量$1 替换为你在命令行上给出的参数值（在脚本程序里，$1、$2

等变量依次对应该脚本的第 1 个、第 2 个命令行参数)。这样，运行这个脚本时，你在命令行上给出的参数值将成为查询命令中的检索关键字。

在运行这个脚本程序之前，还需要把它设置为可执行，如下所示：

```
% chmod +x interests.sh
```

现在，你再也用不着在每次运行这个脚本时都要先编辑了。只需通过命令行参数告诉它你想查找什么东西，就可以得到你想要的资料：

```
% ./interests.sh depression
% ./interests.sh Jefferson
```

可以在sampdb发行版本的misc子目录里找到这个interests.sh脚本,以及与之等价的Windows批处理文件 interests.bat。

说明　我强烈建议大家不要把这类脚本安装在共享区域里，因为它们不进行任何安全方面的检查，因而很容易遭到 SQL 注入攻击。万一有人用如下所示的命令行来调用脚本：

```
% ./interests.sh "Jefferson';DROP DATABASE sampdb;"
```

其后果将是把一条 DROP DATABASE 语句注入到脚本语句中成为 mysql 工具程序的输入，并真的会被执行。

1.6　后面各章的学习计划

通过本章的学习，相信大家对 MySQL 的使用方法已经有了一定的了解。你们应该掌握的技能包括：创建数据库和数据表，对数据表里的记录用多种方法插入、检索、修改、删除等操作。但本章只介绍了一些最浅显的概念，还有很多内容没有涉及。这一点可以从 sampdb 数据库的现状清楚地反映出来。我们创建了这个数据库和其中的数据表，还把一些原始数据填充到了数据表里。在学习过程中，我们还编写了一些查询命令，利用从数据库检索出来的信息解答了一些问题。但是，仍有很多事情在等着我们去做。例如，截止到目前，还没有一种简便的交互方式可以为考试记分项目插入新的考试分数记录和为“美国历史研究会”增加新的会员；不能方便地对现有数据记录进行修改；还没有生成“美国历史研究会”会员名录的打印版和在线版，等等。这些任务需要我们在今后各章（尤其是第 8 章和第 9 章）的学习过程中逐步完成。

如何开展后面的学习取决于读者对哪部分最感兴趣。如果你最想知道的是如何完成“美国历史研究会”和“考试记分项目”里的各项任务，本书的第二部分对 MySQL 应用程序的编写工作进行了讨论。如果你打算朝着 MySQL 数据库管理员的方向努力，本书的第三部分对管理工作进行了探讨。不过，我建议大家还是先按部就班地学完第一部分，多积累一些 MySQL 在使用方面的背景知识比较好。这些内容将帮助大家进一步了解 SQL 语句的语法和用法，明白 MySQL 怎样处理数据，怎样才能让查询执行得更快。对这些内容的扎实掌握将使你有能力胜任与 MySQL 有关的任何工作——无论是使用 mysql 程序编写自己的程序，还是作为一名称职的 MySQL 数据库管理员。

第 2 章

使用 SQL 管理数据

SQL（Structured Query Language，结构化查询语言）是 MySQL 服务器能够听懂的语言，是我们用来告诉 MySQL 服务器如何完成各种数据管理操作的手段。因此，要想有效地与 MySQL 服务器交流，就必须熟练掌握 SQL 语言。当你使用某个程序（如 mysql 客户工具）的时候，它在本质上只是一种能够让你把想要执行的 SQL 语句发送到服务器去的工具而已。如果你使用某种具备 MySQL 编程接口（如 Perl DBI 模块或 PHP PDO 扩展）的语言编写程序，你将能够通过发出 SQL 语句去与服务器进行交流。

第 1 章对 MySQL 的许多方面进行了简要的介绍，其中已经包括了 SQL 的一些基本用法。现在，我们将在这基础上从以下几个方面对 MySQL 所实现的 SQL 语言进行更详细的探讨：

- 改变SQL模式以影响MySQL服务器的行为；
- 各种数据库元素的命名规则；
- 使用多种字符集；
- 数据库、数据表和索引的创建和销毁；
- 获得关于数据库及其内容的信息；
- 使用联结、子查询和联合操作去检索数据；
- 创建视图以便从不同的角度去查看数据表里的数据；
- 多个数据表的删除和刷新操作；
- 利用事务处理机制一次性执行或撤销多条语句；
- 创建外键关系；
- 使用FULLTEXT搜索引擎。

上面列出的项目覆盖了 SQL 语言的众多应用领域。其他章节提供了更多与 SQL 相关的信息。

- 第4章讨论如何创建和使用存储函数（stored function）、存储过程（stored procedure）、触发器和事件。
- 第12章描述如何使用系统管理类语句如GRANT和REVOKE去管理用户账户。这一章还将讨论MySQL数据库的权限控制子系统，它控制着各个账户都允许执行哪些操作。
- 附录E列出了MySQL所实现的各种SQL语句的语法。它还讨论了在SQL语句中使用注释的语法。

你还可以参考《MySQL 参考手册》，它对了解 MySQL 最新版本中的更新非常有用。

2.1 MySQL 服务器的 SQL 模式

MySQL 服务器有一个名为 sql_mode 的系统变量可以让你调控其 SQL 模式，SQL 模式对 SQL 语

句的执行情况有多方面的影响。对这个变量可以作出全局性的设置，而各个客户可以通过改变这个模式来影响它们自己对 MySQL 服务器的连接。这意味着任何一个客户都可以在不影响其他客户的前提下改变 MySQL 服务器对自己的反应。

受 SQL 模式影响的行为包括在数据录入阶段如何处理非法数据、如何引用各种标识符，等等。下面的列表描述了几种可能的 SQL 模式设置值。

- STRICT_ALL_TABLES和STRICT_TRANS_TABLES将启用严格模式。在严格模式下，MySQL服务器在接受“坏”数据值方面将更加严格。（具体地说，它将拒绝“坏”数据值而不是把它们转换为最接近的合法值。）
- TRADITIONAL是另一种复合模式。它类似于严格模式，但启用了其他几种引入额外限制条件的模式以进行更加严格的数据检查。TRADITIONAL模式将导致MySQL服务器在处理“坏”数据值时更接近于传统的SQL服务器。
- ANSI_QUOTES告诉MySQL服务器把双引号识别为一个标识符引用字符。
- PIPES_AS_CONCAT将导致“||”字符串被视为一个标准的SQL字符串合并操作符，而不是“OR”操作符的一个同义词。
- ANSI是一种复合模式。它将同时启用ANSI_QUOTES、PIPES_AS_CONCAT和另外几种模式值，其结果是让MySQL服务器的行为比它的默认运行状态更接近于标准的SQL。

3.3 节将集中讨论会对数据录入环节中的错误或缺失值的处理行为产生影响的 SQL 模式值。附录 D 将对 sql_mode 变量的可取模式值做全面的描述。

在设置 SQL 模式时，需要给出一个由单个模式值或多个以逗号分隔的模式值构成的值；或者，给出一个空字符串以清除该值。模式值不区分字母的大小写。

如果想在启动 MySQL 服务器的时候设置 SQL 模式，可以在 mysqld 命令行上或是在某个选项文件里使用 sql_mode 选项。在命令行上，可以使用一个如下所示的设置项：

```
--sql-mode="TRADITIONAL"
--sql-mode="ANSI_QUOTES,PIPES_AS_CONCAT"
```

如果想在运行时改变 SQL 模式，可以使用一条 SET 语句来设置 sql_mode 系统变量。

任何一个客户都可以给它自己设置一个本次会话专用的 SQL 模式：

```
SET sql_mode = 'TRADITIONAL';
```

如果需要对 SQL 模式作全局性设置，需要加上 GLOBAL 关键字：

```
SET GLOBAL sql_mode = 'TRADITIONAL';
```

设置全局变量需要具备 SUPER 管理权限，新设置值将成为此后连接的所有客户的默认 SQL 模式。

如果想知道会话级或全局级 SQL 模式的当前值，可以使用如下所示的语句：

```
SELECT @@SESSION.sql_mode;
SELECT @@GLOBAL.sql_mode;
```

上述语句的返回值是由当前启用的所有模式以逗号分隔而构成的一列值。如果当前没有启用任何模式，则返回一个空值。

关于用户权限和设置/查看系统变量的其他信息，请参阅第 12 章。

2.2　MySQL 标识符语法和命名规则

几乎所有的 SQL 语句都需要以某种方式使用标识符来引用某个数据库或数据库所容纳的元素，

如数据表、视图、数据列、索引、存储例程、触发器或事件。在引用数据库的元素时，标识符必须遵守以下规则。

标识符里的合法字符。不加引号的标识符必须由系统字符集（utf8）中的字母和数字字符，再加上“_”和“$”字符构成。标识符的第一个字符可以是允许用在标识符里的任何一种字符，包括数字。不过，不加引号的标识符不允许完全由数字字符构成，因为那会使它与数值难以区分。MySQL 允许标识符以数字字符开头的做法在种类繁多的数据库系统中是不常见的。如果打算使用一个这样的标识符，必须特别留意它是否还包含着一个“E”或“e”字符，因为那样的组合很容易导致表达式出现歧义。比如说，表达式“23e + 14”（“+”号两边有空格）意味着给数据列 23e 加上 14，可是“23e+14”又该如何解释呢？它是意味着同样的值，还是一个用科学计数法表示的数值呢？

标识符可以用反引号字符（“`”）括起来（加以界定），这意味着允许使用任意字符，只有取值为 0 或 255 的单字节例外：

```
CREATE TABLE `my table` (`my-int-column` INT);
```

当标识符是一个 SQL 保留字或者包含空格或其他特殊字符的时候，给它加上引号非常实用。给标识符加上引号让它可以完全由数字字符构成，这对不加引号的标识符来说是不允许的。如果想在一个加上引号的标识符里使用一个标识符引号字符，双写它即可。

在 MySQL 5.1.6 版之前，用于数据库和数据表的标识符还必须遵守另外两个限制条件，即使它们已经加上了引号。其一，不允许使用“.”字符，因为该字符在 `db_name.tbl_name` 或 `db_name.tbl_name.col_name` 格式的名称里被用作分隔符。其二，不允许使用 Unix 或 Windows 的路径名分隔字符（即“/”或“\”）。之所以不允许在数据库和数据表标识符里使用路径名分隔符，是因为数据库在硬盘上被表示为子目录，数据表在硬盘上被表示为至少一个文件。既然如此，这类标识符就只能由允许用在子目录名和文件名里的合法字符构成。不允许把 Unix 路径名分隔符用在 Windows 平台上（反之亦然），是为了让在运行于不同平台上的 MySQL 服务器之间迁移数据库和数据表的工作更容易。（如果允许人们在 Windows 平台上的数据表名字里使用斜线字符，就无法把它迁移到 Unix 平台上了，因为后一种平台上的文件名不允许包含斜线字符。）

从 MySQL 5.1.6 版开始，把 SQL 语句里的标识符映射为目录名和文件名的机制经过了修改，使得早期版本里的部分非法字符也可以用在标识符里。具体地说，只需给标识符加上引号，在其中使用路径名字符（“/”或“\”）和“.”字符就是合法的了。

你的操作系统可能会对数据库和数据表标识符有额外的要求，请参阅 11.2.6 节。

数据列和数据表名字的假名可以相当随意。如果打算使用的假名是一个 SQL 保留字、完全由数字构成，或者包含空格或其他特殊字符，就应该用标识符引号字符把它括起来。数据列假名还可以使用单引号或双引号。

MySQL 服务器的 SQL 模式。如果启用了 `ANSI_QUOTES` SQL 模式，可以用双引号来括住标识符（反引号仍允许使用）。

```
CREATE TABLE "my table" ("my-int-column" INT);
```

注意 启用 `ANSI_QUOTES` 还有额外效果——字符串值必须用单引号写出。如果使用了双引号，MySQL 服务器将把该值解释为标识符而不是字符串。

内建函数的名字一般来说都不是保留字，可以不加引号地用作标识符。不过，如果启用了 IGNORE_SPACE SQL 模式，函数名将被视为保留字，还想使用它们作为标识符就必须给它们加上引号。设置 SQL 模式的具体步骤请参阅 2.1 节。

标识符的长度。绝大多数标识符的最大长度是 64 个字符。假名的最大长度是 256 个字符。

标识符限定符。根据上下文，标识符可能需要加以限定，以明确它到底对应着什么。如果想指称一个数据库，把它的名字写出来即可：

```
USE db_name;
SHOW TABLES FROM db_name;
```

如果想指称一个数据表，有两种选择。

- 使用完整的数据表名，它由一个数据库标识符和一个数据表标识符构成：

```
SHOW COLUMNS FROM db_name.tbl_name;
SELECT * FROM db_name.tbl_name;
```

- 一个数据表标识符本身对应着默认（当前）数据库里的一个数据表。如果sampdb是默认数据库，下面的语句是等效的：

```
SELECT * FROM member;
SELECT * FROM sampdb.member;
```

如果没有选定数据库，就不能在没有给出数据库限定符的情况下引用某个数据表，因为这个数据表到底属于哪个数据库是不明确的。

对数据表名加以限定的考虑同样适用于视图（它们是“虚拟的”数据表）和存储程序的名字。

如果想指称一个数据列，有 3 种选择。

- 使用完整的数据列名，如*db_name.tbl_name.col_name*。
- 对于默认数据库里某给定数据表里的一个数据列，可以使用*tbl_name.col_name*形式的部分限定名。
- 只写出一个非完整名*col_name*表示该数据列属于上下文环境所确定的那个数据表。下面两个查询使用了相同的数据列名，但每条语句的FROM子句所提供的上下文表明了应该到哪一个数据表去选择这些数据列：

```
SELECT last_name, first_name FROM president;
SELECT last_name, first_name FROM member;
```

一般来说，没有必要提供完整的名字，但如果你愿意，那永远都是合法的。如果用一条 USE 语句选定了一个数据库，该数据库就将成为默认数据库并隐含在你此后引用的每一个不完整的数据表名字里。如果在一条 SELECT 语句里只引用了一个数据表，该数据表隐含在这条语句所包含的每一个数据列名字里。只有在无法根据上下文确定数据表或数据库的时候才必须使用完整的标识符。比如说，如果一条语句涉及多个数据库里的数据表，所有不在默认数据库里的数据表就必须以 *db_name.tbl_name* 的形式来给出，这样才能让 MySQL 知道哪个数据库包含着哪个数据表。同样的道理，如果某个查询涉及多个数据表并引用了一个在多个数据表里用到的数据列名字，就必须用一个数据表标识符对该数据列标识符进行限定，以明确打算使用的是哪一个数据列。

如果打算在引用一个完整名字时使用引号，就应该给该名字里的每一个标识符分别加上引号。如下所示：

```
SELECT * FROM `sampdb`.`member` WHERE `sampdb`.`member`.`member_id` > 100;
```

不要把这样的名字作为一个整体而只加上一组引号。下面这条语句是不正确的：

```
SELECT * FROM `sampdb.member` WHERE `sampdb.member.member_id` > 100;
```

在把某个保留字用作标识符时，必须给它加上引号，但在这个保留字紧跟在一个句号限定符后面时例外，因为已经可以根据上下文而确定这个保留字其实是一个标识符。

2.3 SQL 语句中的字母大小写问题

SQL 语句中的字母大小写规则随语句元素的不同而变化，同时还要取决于你正引用的事物和 MySQL 服务器主机上的操作系统。

SQL 关键字和函数名。关键字和函数名不区分字母的大小写。它们可以为任意的字母大小写组合。下面的语句将检索出同样的信息（但输出结果中的数据列标题将是不同的字母大小写组合）：

```
SELECT NOW();
select now();
sElEcT nOw();
```

数据库、数据表和视图的名字。在服务器主机上，MySQL 数据库和数据表用底层文件系统中的目录和文件表示。因而数据库和数据表名字的默认字母大小写情况将取决于服务器主机上的操作系统在文件名方面的规定。Windows 文件名不区分字母的大小写，所以运行在 Windows 主机上的 MySQL 服务器也就不区分数据库和数据表名字的字母大小写。运行在 Unix 主机上的 MySQL 服务器往往要区分数据库和数据表名字的字母大小写，因为 Unix 文件系统是区分字母大小写的。Mac OS X 平台上的 HFS+文件系统中的名字是个例外，不区分字母的大小写。

MySQL 使用一个文件来表示一个视图，所以刚才与数据表有关的讨论也同样适用于视图。

存储程序的名字。存储函数、存储过程和事件的名字不区分字母的大小写。触发器的名字要区分字母的大小写，这一点和标准的 SQL 行为是不一样的。

数据列和索引的名字。数据列和索引的名字在 MySQL 环境里不区分字母的大小写。下面的语句将检索出同样的信息：

```
SELECT name FROM student;
SELECT NAME FROM student;
SELECT nAmE FROM student;
```

别名的名字。在默认的情况下，数据表别名区分字母的大小写。可以使用任意的字母大小写组合（大写、小写或大小写混用）来给出一个别名，但如果需要在同一条语句里多次用到同一个别名，就必须让它们保持同样的字母大小写组合。如果 `lower_case_table_names` 变量是非零值，数据表别名将不区分字母的大小写。

字符串值。字符串值是否区分字母大小写，这取决于它是二进制还是非二进制，而非二进制字符串还要取决于字符集的排序方式。这对字符串常数值和字符串数据列的内容都不例外。关于这方面的更多信息请参阅 3.1.2 节。

当你在一台区分文件名字母大小写的机器上创建数据库和数据表时，应该这样思考字母大小写的问题：日后有没有可能需要把它们迁移到一台不区分文件名字母大小写的机器上去？假设你已经在一台 Unix 服务器上创建了两个名字分别是 abc 和 ABC 的数据表——这两个名字在这台服务器上是区别对待的，当你想把这两个数据表迁移到一台 Windows 机器上时就会遇到麻烦。因为新机器不区分字母的大小写，abc 和 ABC 将无法区别对待。在把数据表从一台 Unix 主服务器复制到一台 Windows 从服

务器时也会遇到麻烦。

避免字母大小写问题演变成棘手难题的办法之一，是选定一种字母大小写方案，一直遵照该方案去创建数据库和数据表。这样一来，等你日后想把某个数据库迁移到不同的服务器上时，名字的大小写问题就不存在了。我的建议是统一使用小写字母。这在你使用 InnoDB 数据表时也有益处，因为 InnoDB 引擎在其内部是把数据库和数据表的名字保存为小写字母的。

如果想统一使用小写字母来创建数据库和数据表的名字——就算没在 CREATE 语句里特意设定也能如此，可以通过设置 lower_case_table_name 系统变量来配置服务器。更多信息请参阅 11.2.6 节。

不管你的系统是否区分数据库或数据表名字的字母大小写，你在给定的查询里必须使用一致的大小写组合来引用它。SQL 关键字、函数名、数据列名和索引名不必如此拘泥，因为 MySQL 允许在查询命令里使用任意的字母大小写组合。不过，如果能坚持使用统一的风格而不是随意混搭的话，查询命令会有更好的可读性。

2.4　字符集支持

MySQL 不仅支持多种字符集，还允许对服务器、数据库、数据表、数据列或字符串常数级别的字符集作出互不影响的设定。比如说，如果想让某个数据表的数据列默认使用 latin1 字符集，但同时还包含一个 Hebrew 数据列和一个 Greek 数据列，你完全可以那样做。此外，还可以明确地设定排序方式。有哪些字符集和排序方式可供选择是可以查出来的，把数据从一种字符集转换为另一种也有章可循。

本节提供了关于 MySQL 的字符集支持的基本背景知识。第 3 章将更细致地讨论字符集、排序方式、二进制字符串和非二进制字符串以及如何定义和使用基于字符的数据表列。第 12 章将讨论如何对 MySQL 服务器支持的字符集进行配置。

MySQL 的字符集支持机制提供了以下一些功能。

- MySQL 服务器允许同时使用多种字符集。
- 一种给定的字符集可以有一种或多种排序方式。你可以为应用程序挑选一种最适用的排序方式。
- Unicode 支持由 utf8 和 ucs2 字符集提供，从 MySQL 6.0.4 版开始有更多的字符集可供选用。
- 你可以在服务器、数据库、数据表、数据列和字符串常数等级别设定字符集：
  - 服务器有一个默认的字符集。
  - CREATE DATABASE 语句可以用来设定数据库级字符集，ALTER DATABASE 语句可以改变之。
  - CREATE TABLE 和 ALTER TABLE 语句有专门用来设定数据表级和数据列级的子句（详见第 3 章）。
  - 用于字符串常数的字符集既可以通过上下文设定，也可以明确设定。
- 既有用来转换数据值的字符集的函数和操作符，也有用来判断数据值的字符集的函数和操作符。类似地，COLLATE 操作符可以用来改变某个字符串的排序方式，而 COLLATE() 函数将返回某给定字符串的排序方式。
- SHOW 语句和 INFORMATION_SCHEMA 数据表提供了关于可用字符集和排序方式的信息。
- 当你改变某个带索引的字符类型的数据列时，MySQL 服务器将自动地对索引进行重新排序。

在同一个字符串内不能混用不同的字符集，一个给定的数据列不能在不同的数据行使用不同的字

符集。不过，可以选用一种 Unicode 字符集（用单一编码方案表示多种语言的字符）去实现你期望的多语言支持。

2.4.1 字符集的设定

字符集和排序方式可以在多个级别进行设定，从 MySQL 服务器使用的默认字符集到用于个别字符串的字符集。

服务器的默认字符集和排序方式是在当初编译时内建的，但我们可以在启动服务器时使用 --character-set-server 和--collation-server 选项，或是在服务器启动后设置 character-set-server 和 collation-server 系统变量来覆盖它们。如果只选定了字符集，它的默认排序方式就将成为服务器的默认排序方式。如果想选定一种排序方式，必须让它与字符集保持兼容。（判断排序方式与字符集是否兼容的办法，是看它的名字是否以字符集的名字开头。比如说，utf8_danish_ci 排序方式与 utf8 字符集相兼容、与 latin1 字符集不兼容。）

在创建数据库和数据表的 SQL 语句里，有两个子句专门用来设定数据库、数据表和数据列级别的字符集和排序方式：

```
CHARACTER SET charset
COLLATE collation
```

CHARSET 可以用作 CHARACTER SET 的同义词。*charset* 是服务器所支持的字符集的名字。而 *collation* 是该字符集的一种排序方式的名字。这些子句可以同时使用，也可以分开使用。在同时使用这两个子句的时候，必须让排序方式的名字与字符集保持兼容。如果只给出了 CHARACTER SET 子句，则意味着使用默认排序方式。如果只给出了 COLLATE 子句，则使用由给定排序方式的名字的开头部分确定的字符集。这些规则适用于以下几个级别。

- 如果想在创建数据库时为它设定一个默认的字符集和排序方式，需要使用如下所示的语句：

  ```
  CREATE DATABASE db_name CHARACTER SET charset COLLATE collation;
  ```

  如果没有对字符集或排序方式作出设定，服务器级别的默认设置将传递给这个数据库。
- 如果想为某个数据表设定默认的字符集和排序方式，可以在创建该数据表时利用CHARACTER SET和COLLATION数据表选项：

  ```
  CREATE TABLE tbl_name (...) CHARACTER SET charset COLLATE collation;
  ```

  如果没有对字符集或排序方式作出设定，数据库级别的默认设置将传递给这个数据表。
- 对于数据表里的某个数据列，可以使用CHARACTER SET和COLLATION属性为它指定一个字符集和排序方式。如下所示：

  ```
  c CHAR(10) CHARACTER SET charset COLLATE collation
  ```

  如果没有对字符集或排序方式作出设定，数据表级别的默认设置将传递给这个数据列。这些属性适用于 CHAR、VARCHAR、TEXT、ENUM 和 SET 数据类型。

还可以利用 COLLATE 操作符按照特定排序方式对字符串值排序。比如说，假设 c 是一个使用 latin1 字符集、latin1_swedish_ci 排序方式的数据列，但你想按照 Spanish 排序规则对它排序，可以这么做：

```
SELECT c FROM t ORDER BY c COLLATE latin1_spanish_ci;
```

2.4.2　确定可供选用的字符集和当前设置

如果想知道有哪些字符集和排序方式可供选用，可用使用下面这些语句：

```
SHOW CHARACTER SET;
SHOW COLLATION;
```

这两条语句都支持使用一个 LIKE 子句，把查询结果缩窄到名字与给定模式相匹配的那些字符集或排序方式。比如说，下面这条语句将只列出基于拉丁语的字符集：

```
mysql> SHOW CHARACTER SET LIKE 'latin%';
+---------+-----------------------------+-------------------+--------+
| Charset | Description                 | Default collation | Maxlen |
+---------+-----------------------------+-------------------+--------+
| latin1  | cp1252 West European        | latin1_swedish_ci |      1 |
| latin2  | ISO 8859-2 Central European | latin2_general_ci |      1 |
| latin5  | ISO 8859-9 Turkish          | latin5_turkish_ci |      1 |
| latin7  | ISO 8859-13 Baltic          | latin7_general_ci |      1 |
+---------+-----------------------------+-------------------+--------+
```

下面这条语句将只列出与 utf8 字符集相兼容的排序方式（排序方式的名字总是以字符集的名字开头）：

```
mysql> SHOW COLLATION LIKE 'utf8%';
+--------------------+---------+-----+---------+----------+---------+
| Collation          | Charset | Id  | Default | Compiled | Sortlen |
+--------------------+---------+-----+---------+----------+---------+
| utf8_general_ci    | utf8    |  33 | Yes     | Yes      |       1 |
| utf8_bin           | utf8    |  83 |         | Yes      |       1 |
| utf8_unicode_ci    | utf8    | 192 |         | Yes      |       8 |
| utf8_icelandic_ci  | utf8    | 193 |         | Yes      |       8 |
| utf8_latvian_ci    | utf8    | 194 |         | Yes      |       8 |
| utf8_romanian_ci   | utf8    | 195 |         | Yes      |       8 |
| utf8_slovenian_ci  | utf8    | 196 |         | Yes      |       8 |
...
```

从上面这些语句的输出结果里可以看出，每一种字符集最少拥有一种排序方式，并且有一种是它的默认排序方式。

关于可用字符集和排序方式的信息，还可以从 INFORMATION_SCHEMA 数据库中的 CHARACTER_SETS 和 COLLATIONS 数据表查到（请参阅 2.7 节）。

如果想显示 MySQL 服务器的当前字符集和排序方式的设置情况，可以使用 SHOW VARIABLES 语句：

```
mysql> SHOW VARIABLES LIKE 'character\_set\_%';
+--------------------------+--------+
| Variable_name            | Value  |
+--------------------------+--------+
| character_set_client     | latin1 |
| character_set_connection | latin1 |
| character_set_database   | latin1 |
| character_set_filesystem | binary |
| character_set_results    | latin1 |
| character_set_server     | latin1 |
```

```
| character_set_system     | utf8   |
+--------------------------+--------+
mysql> SHOW VARIABLES LIKE 'collation\_%';
+----------------------+-------------------+
| Variable_name        | Value             |
+----------------------+-------------------+
| collation_connection | latin1_swedish_ci |
| collation_database   | latin1_swedish_ci |
| collation_server     | latin1_swedish_ci |
+----------------------+-------------------+
```

这些系统变量中的某几个会对客户在与 MySQL 服务器建立连接后的通信情况产生影响。这方面的细节请参阅 3.1.2 节的第 2 小节。

2.4.3 Unicode支持

之所以会有这么多种字符集，原因之一是人们为不同的人类语言制定了不同的字符编码方案。这就导致了几个问题。比如说，如果某给定字符在好几种人类语言里都存在，它在不同的编码方案里就有可能是用不同的数值来表示的。还有，不同的人类语言往往需要使用数目不同的字节去表示一个字符。`latin1` 字符集足够小，每个字符只需使用一个字节来表示，但有些语言（如日语和汉语）因为包含非常多的字符，它们需要使用多个字节来表示每个字符。

Unicode 的目标是提供一个统一的字符编码方案，让所有人类语言的字符集都能以一种统一的方式表示。

1. MySQL 6.0版之前的Unicode支持

在 MySQL 6.0.4 版之前，其 Unicode 支持仅包括 Basic Multilingual Plane（BMP，初级多语言方案）里的字符，最多只有 65 536 个字符。没被收录到 BMP 方案里的其他字符是没有任何支持的。Unicode 通过两种字符集提供了一个比较完善的解决方案。

- `ucs2`字符集对应着Unicode UCS-2编码方案。它使用两个字节来表示一个字符，高位字节排列在前。这种字符集无法表示需要用两个以上的字节才能表示的字符。UCS是Universal Character Set（通用字符集）的缩写。
- `utf8`字符集采用了一种长度可变的格式，使用一到三个字节来表示一个字符。它对应着UTF-8编码方案。UTF是Unicode Transformation Format（统一编码转换格式）的缩写。

2. MySQL 6.0版之后的Unicode支持

从 MySQL 6.0.4 版开始，其 Unicode 支持把 BMP 方案所遗漏的补充字符也收录了进来，这么做的效果如下所示。

- `ucs2`字符集在MySQL 6.0系列版本里未做改动，每个字符仍占两个字节。新增加的`utf16`和`utf32`字符集类似于`utf8`，但扩充了对补充字符的支持。在`utf16`字符集里，BMP字符仍占两个字节（和usc2字符集一样），补充字符占四个字节。在`utf32`字符集里，所有字符都占四个字节。
- 以前，每个`utf8`字符占一到三个字节。增加了补充字符之后，每个`utf8`字符占一到四个字节。
- 由MySQL 6.0以前的版本创建的、使用`utf8`字符集的数据库和数据表在MySQL 6.0系列版本里将按照`utf8mb3`字符集来显示。（比如说，如果使用`SHOW CREATE TABLE`语句去查看的话，你将看到`utf8mb3`。）除了名字本身的差异，MySQL 6.0系列版本中的`utf8mb3`字符集和6.0系列

之前的utf8字符集完全一样。

如果想把数据表从老 utf8 字符集（3 字节）转化为新 utf8（4 字节），可以在升级到 MySQL 6.0 以前先用 mysqldump 工具把数据表备份下来，等升级完后再重新加载导出文件。在升级完成以后，千万不要忘记运行 mysql_upgrade 工具以确保 mysql 数据库里的系统级数据表全都打好了补丁。

2.5 数据库的选定、创建、删除和变更

MySQL 提供了几个数据库级的语句：USE 用来选定一个默认数据库，CREATE DATABASE 用来创建数据库，DROP DATABASE 用来删除数据库，ALTER DATABASE 用来改变数据库的全局特性。

在后一种情况的语句中，关键字 SCHEMA 是 DATABASE 的同义词。

2.5.1 数据库的选定

USE 语句选定一个数据库并把它当做指定 MySQL 服务器连接上的默认（当前）数据库：

```
USE db_name;
```

要想选定一个数据库，就必须具备相应的访问权限，要不然无法选定它。

显式选定数据库不是必要的。如果你确实有访问该数据库的权限，那么即使你没有选择数据库，只要用数据库名字来限定数据表名字，你就可以使用其中的数据表。比如说，如果你没有事先选定 sampdb 数据库，却想检索其中 president 数据表里的内容，可以使用如下所示的查询语句：

```
SELECT * FROM sampdb.president;
```

不过，通常不带数据库限定词的数据表名字用起来要更方便一些。

选定默认数据库并不意味着它将在连接持续期间内一直是默认数据库。只要具备足够的访问权限，你可以多次使用 USE 语句在多个数据库之间任意切换。同时，选定一个数据库也并不意味着只能使用这个数据库里的数据表。即使已经把某个数据库选定为当前的默认数据库，也可以利用数据库标识符通过名字去访问其他数据库里的数据表。

当与服务器的连接终止时，该服务器上的默认数据库概念也就不复存在了。换句话说，当你再次连接上该服务器时，它并不会记得你上一次选定的默认数据库。

2.5.2 数据库的创建

要创建一个数据库，需要使用 CREATE DATABASE 语句：

```
CREATE DATABASE db_name;
```

执行数据库创建操作的先决条件是：数据库名字必须是合法的，这个数据库不能是已经存在的，你必须有足够的权限去创建它。

创建数据库时，MySQL 服务器会在它的数据目录里创建一个与该数据库同名的子目录，这个新目录称为数据库子目录。服务器还会在那个数据库目录里创建一个 db.opt 文件来保存数据库的属性。

CREATE DATABASE 语句有好几种可选的子句。它的完整语法如下所示：

```
CREATE DATABASE [IF NOT EXISTS] db_name
  [CHARACTER SET charset] [COLLATE collation];
```

在正常情况下，当你试图创建的数据库已经存在时，系统将报告出错。如果想避免这类错误，只在给定数据库尚不存在的前提下才创建它，请加上一条 IF NOT EXISTS 子句：

```
CREATE DATABASE IF NOT EXISTS db_name;
```

在默认情况下，服务器级别的字符集和排序方式将成为新建数据库的默认字符集和排序方式。可以使用 CHARACTER SET 和 COLLATE 子句对这些数据库属性作出明确的设置。如下所示：

```
CREATE DATABASE mydb CHARACTER SET utf8 COLLATE utf8_icelandic_ci;
```

如果只给出了 CHARACTER SET 子句而没有 COLLATE 子句，则意味着使用给定字符集的默认排序方式。如果只给出了 COLLATE 子句而没有 CHARACTER SET 子句，则意味着使用排序方式的名字的开头部分确定的字符集。

字符集必须是服务器所支持的，如 latin1 或 sjis。排序方式必须是给定字符集的一个合法的排序方式。关于字符集和排序方式的进一步讨论请参阅第 3 章。

MySQL 把数据库的字符集和排序方式等属性保存在相应的 db.opt 文件里。在创建新数据表时，如果你没有在新数据表的定义里为它指定一种默认的字符集和排序方式，数据库级别的默认设置就将成为新数据表的默认设置。

如果想查看现有数据库的定义，可以使用一条 SHOW CREATE DATABASE 语句：

```
mysql> SHOW CREATE DATABASE mydb\G
*************************** 1. row ***************************
       Database: mydb
Create Database: CREATE DATABASE `mydb`
                 /*!40100 DEFAULT CHARACTER SET utf8
                 COLLATE utf8_icelandic_ci */
```

2.5.3　数据库的删除

只要你有足够的权限，删除一个数据库和创建一个数据库同样简单，使用如下语句即可：

```
DROP DATABASE db_name;
```

注意，千万不要随意使用 DROP DATABASE 语句，这条语句将会删掉数据库和其中的所有东西，包括数据表、存储例程等，这个数据库也就永远消失了，除非定期地对数据库进行备份。

一个数据库就是 MySQL 数据目录里的一个子目录，这个子目录用于存放数据表视图和触发器等。如果 DROP DATABASE 语句失效，通常是因为那个数据库子目录里还包含有一些与数据库对象无关的文件。DROP DATABASE 语句不会删除这类文件，因而也就不删除那个数据库子目录。这就意味着如果写了 SHOW DATABASES 语句，尽管里面已经没有数据表了，可那个数据库子目录却依然存在。在这种情况下，如果真想删除那个数据库，就必须手动删除该数据库子目录里遗留的文件和子目录本身，然后再发出 DROP DATABASE 语句。

2.5.4　数据库的变更

使用 ALTER DATABASE 语句可以改变数据库的全局特性。就目前而言，数据库的全局特性还只有默认字符集和排序规则：

```
ALTER DATABASE [db_name] [CHARACTER SET charset] [COLLATE collation];
```

2.6　数据表的创建、删除、索引和变更

MySQL 允许创建、删除数据表或改变其结构，相应的 SQL 语句分别是 CREATE TABLE、DROP TABLE 和 ALTER TABLE。CREATE INDEX 和 DROP INDEX 语句用来给现有的数据表增加或删除索引。以下几节将详细解释这些语句，但我认为应该先讨论一下 MySQL 为管理不同类型的数据表而支持的几种存储引擎。

2.6.1　存储引擎的特征

MySQL 支持好几种*存储引擎*（storage engine，它们以前被称为“数据表处理器”）。由同一个存储引擎所实现的数据表具有一些共同的属性或特征。表 2-1 简要地描述了 MySQL 发行版本目前支持的几种存储引擎，稍后将讨论各存储引擎的功能细节。在 MySQL 5.0 和更高的版本里，表中除 Falcon 以外的所有存储引擎都可以直接使用，Falcon 存储引擎只能在 MySQL 6.0 里使用。

表 2-1　MySQL 支持的存储引擎

存储引擎	说　明
ARCHIVE	用于数据存档的引擎（数据行被插入后就不能再修改了）
BLACKHOLE	这种存储引擎的写操作是删除数据，读操作是返回空白记录
CSV	这种存储引擎在存储数据时以逗号作为数据项之间的分隔符
EXAMPLE	示例（存根）存储引擎
Falcon	用来进行事务处理的存储引擎
FEDERATED	用来访问远程数据表的存储引擎
InnoDB	具备外键支持功能的事务处理引擎
MEMORY	内存里的数据表
MERGE	用来管理由多个 MyISAM 数据表构成的数据表集合
MyISAM	默认的存储引擎
NDB	MySQL Cluster 专用存储引擎

有几种存储引擎还有同义名称，MRG_MyISAM 和 NDBCLUSTER 分别是 MERGE 和 NDB 的同义名称。MEMORY 和 InnoDB 存储引擎最早分别叫做 HEAP 和 Innobase，后者现在仍允许使用，但已不再建议使用。

在 MySQL 5.1 和更高的版本里，服务器采用了一种“可插入”的体系结构，它提供一套标准的接口用来在运行时动态地加载和卸载存储引擎。因此，由第三方开发的存储引擎可以被方便地集成到服务器里。

1. 查看有哪些存储引擎可供选用

给定一个服务器，真正使用哪几种存储引擎将取决于你的 MySQL 版本、在编译该服务器时使用的具体配置、启动该服务器时使用的选项，等等。配置和启动存储引擎的具体步骤参见 12.7 小节。

用 SHOW ENGINES 语句可以查出服务器都知道哪些存储引擎。该语句提供的信息可以帮助我们回答诸如“有哪些支持事务处理的存储引擎可供选用”之类的问题。以下输出内容使用的是 MySQL 5.0 版的格式：

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
 Engine: MyISAM
Support: DEFAULT
Comment: Default engine as of MySQL 3.23 with great performance
*************************** 2. row ***************************
 Engine: MEMORY
Support: YES
Comment: Hash based, stored in memory, useful for temporary tables
*************************** 3. row ***************************
 Engine: InnoDB
Support: YES
Comment: Supports transactions, row-level locking, and foreign keys
...
```

Support 栏里的 YES 或 NO 代表该存储引擎是否可用，DISABLED 的意思是该存储引擎可用但它已被关闭，DEFAULT 表示这是服务器默认使用的存储引擎。一般来说，DEFAULT 存储引擎应该是可用的。

MySQL 5.1 版里的 SHOW ENGINES 语句要比 5.0 系列版本多几个与事务处理有关的数据列：

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
      Engine: InnoDB
     Support: YES
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
...
*************************** 8. row ***************************
      Engine: MyISAM
     Support: DEFAULT
     Comment: Default engine as of MySQL 3.23 with great performance
Transactions: NO
          XA: NO
  Savepoints: NO
...
```

Transaction 栏里的值表明存储引擎是否支持事务。XA 和 Savepoints 栏里的值表明某种存储引擎是否支持分布式事务处理（本书未讨论）和部分事务回滚。

在 MySQL 5.1 和更高的版本里，有一个名为 ENGINES 的 INFORMATION_SCHEMA 数据表，它提供的信息与 SHOW ENGINES 语句完全一样。你可以像下面这样使用该数据表查看有哪些支持事务处理的存储引擎可供选用（以下输出来自 MySQL 6.0，所以其中包括了 Falcon 存储引擎）：

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.ENGINES
    -> WHERE TRANSACTIONS = 'YES';
+--------+
| ENGINE |
+--------+
| Falcon |
| InnoDB |
+--------+
```

2. 数据表在硬盘上的存储方式

你每创建一个数据表，MySQL 就会创建一个硬盘文件来保存该数据表的格式（也就是它的定义）。这个格式文件的基本名和数据表的名字一样，扩展名是.frm。比如说，如果数据表的名字是 t，其格式文件的名字就将是 t.frm。你创建的数据表属于哪个数据库，服务器就会在该数据库的数据库子目录里创建这个文件。.frm 文件的内容是不变的，不管是哪一个存储引擎在管理数据表，每个数据表也只有一个相应的.frm 文件。如果数据表的名字字符在文件名里会引起麻烦，SQL 语句里使用的数据表的名字有可能与相应的.frm 文件的基本名（表名）不一致。从 SQL 名字到文件名的映射规则参见 11.2.6 节。

具体到某个特定的存储引擎，它还会为数据表再创建几个特定的文件以存储其内容。对于给定的数据表，与之相关的所有文件都集中存放在这个数据表所在的数据库的数据库子目录里。表 2-2 列出了几种存储引擎为特定数据表创建的文件的扩展名。

表2-2　由存储引擎创建的数据表文件

存储引擎	硬盘上的文件
MyISAM	.MYD（数据）、.MYI（索引）
MERGE	.MRG（由各成员MyISAM数据表的名字构成的清单）
InnoDB	.ibd（数据和索引）
ARCHIVE	.ARZ（数据）、.ARM（元数据）
CSV	.CSV（数据）、.CSM（元数据）

对某些存储引擎而言，格式文件是与某特定数据表相关联的唯一文件。其他存储引擎会把数据表的内容保存到硬盘上的其他地方，或者使用一个或多个*表空间*（tablespace，由多个数据表所共享的存储区域），如下所示。

- MEMORY 数据表存放在内存里，不占用任何硬盘空间。
- 在默认的情况下，InnoDB 引擎会把数据表的数据和索引存储在它的共享表空间里。也就是说，所有 InnoDB 数据表的内容都集中保存在一个共享存储空间里，而不是与某个特定的数据表相关联的文件里。InnoDB 引擎只在你特意配置它来为每个数据表分别创建一个表空间时才会去创建.ibd 文件。
- Falcon 引擎把数据表的数据和索引保存在表空间文件里。有一个默认的 Falcon 表空间，但你也可以根据自己的需要创建其他表空间。这些表空间中的任何一个都可以容纳多个数据表的内容。
- BLACKHOLE 和 EXAMPLE 存储引擎实际上不存储任何数据，所以它们不需要创建任何文件。
- FEDERATED 引擎用于访问某远程 MySQL 服务器上的数据表，它本身不创建任何文件。
- 在接下来的几节里，我们将有选择地介绍几种 MySQL 存储引擎在功能和行为方面的特点。如果你想知道各存储引擎怎样以物理方式保存数据表，请参阅 11.2.3 节。

3. MyISAM存储引擎

MyISAM 存储引擎是 MySQL 默认使用的存储引擎，如果你没有把你的服务器配置成其他样子的话。下面是它的部分功能。

- MyISAM 存储引擎提供了键压缩功能。它使用某种压缩算法来保存连续的、相似的字符串索引值。此外，MyISAM 存储引擎还可以压缩相似的数值索引值，因为数值都是按照高位字节优先的办法来保存的。（低位字节的索引值的检索速度非常快，所以高位字节很容易压缩。）

如果你想激活数值压缩功能，请在创建 MyISAM 数据表时使用 PACK_KEYS=1 选项。

- 与其他存储引擎相比，MyISAM 存储引擎为 AUTO_INCREATMENT 数据列提供了更多的功能。关于这方面的详情请参见 3.4 节。
- 每个 MyISAM 数据表都有一个标志，服务器或 myisamchk 程序在检查 MyISAM 数据表时会对这个标志进行设置。MyISAM 数据表还有一个标志用来表明该数据表在上次使用后是不是被正常地关闭了。如果服务器意外宕机或机器崩溃，这个标志可以用来判断数据表是否需要检查和修复。如果你想让这种检查自动进行，需要在启动服务器时使用 --myisam-recover 选项。这会让服务器在每次打开一个 MyISAM 数据表时自动检查该数据表的标志并进行必要的数据表修复处理。
- MyISAM 存储引擎支持全文检索，但这需要通过 FULLTEXT 索引来实现。
- MyISAM 支持空间数据类型和 SPATIAL 索引。

4. MERGE存储引擎

MERGE 数据表提供了一种把多个 MyISAM 数据表合并为一个逻辑单元的手段。查询一个 MERGE 数据表相当于查询其所有的成员数据表。这种安排的好处之一是可以绕开文件系统对各个 MyISAM 数据表的最大长度的限制。

用来构成 MERGE 数据表的所有数据表必须有同样的结构。这意味着必须为各成员数据表里的数据列定义同样的名字、同样的类型和同样的顺序，索引也必须以同样的办法按同样的顺序定义。我们可以把经过压缩和未经过压缩的数据表混杂在一起而构成一个 MERGE 数据表（myisamchk 程序可以用来创建压缩数据表，请参见附录 F）。

2.6.2 节中的第 5 小节给出了一个例子。另外，分区数据表可以作为除 MERGE 数据表以外的另一种选择，而且其成员不限于 MyISAM 数据表。请参阅 2.6.2 节中的第 6 小节。

5. MEMORY存储引擎

MEMORY 存储引擎把数据表保存在内存里，这些数据表有着长度固定不变的数据行，这两个特点使得数据表的检索速度非常之快。

从某种意义上讲，MEMORY 数据表是临时性的。当服务器掉电时，表中的内容也就消失了——MEMORY 数据表在服务器重启之后仍会存在，只是它们的内容将是一片空白。MERORY 数据表的另一个特点是其内容对其他客户来说是可见的，这与用 CREATE TEMPORARY TABLE 语句创建出来的临时数据表形成了对照。

MERORY 数据表的如下特点使它们比其他类型的数据表更容易处理，所以检索速度非常快。

- 在默认的情况下，MERORY 数据表使用散列索引，利用这种索引进行“相等比较”的速度非常快，但进行“范围比较”的速度就慢多了。因此，散列索引只适合用在使用“=”和“<=>”操作符进行的比较操作里，不适合用在使用“<”或“>”操作符进行的比较操作里。出于同样的考虑，散列索引也不适合用在 ORDER BY 子句里。
- 存储在 MERORY 数据表里的数据行使用的是长度固定不变的格式，以此加快处理速度，这意味着你不能使用 BLOB 和 TEXT 这样的长度可变的数据类型。VARCHAR 是一种长度可变的类型，但因为它在 MySQL 内部被当做一种长度固定不变的 CHAR 类型，所以你可以在 MERORY 数据表里使用 VARCHAR 类型。

如果确实需要使用 MERORY 数据表和“<”、“>”或 BETWEEN 操作符进行某种比较以判断某个值是否落在某个范围内，可以使用 BTREE 索引来加快速度（请参阅 2.6.4 节中的第 2 小节）。

6. InnoDB存储引擎

InnoDB 存储引擎最早是由 Innobase Oy 公司开发的，该公司后来被 Oracle 公司收购。InnoDB 存储引擎有以下几种功能。

- ❑ 支持提交和回滚操作，确保数据在事务处理过程中万无一失。还可以通过创建保存点（savepoint）的办法来实现部分回滚（partial rollback）。
- ❑ 在系统崩溃后可以自动恢复。
- ❑ 外键和引用完整性支持，包括递归式删除和更新。
- ❑ 数据行级别的锁定和多版本共存，这使得 InnoDB 数据表在需要同时进行检索和更新操作的复杂查询里表现出非常好的并发性能。
- ❑ 在默认的情况下，InnoDB 存储引擎会把数据表集中存储在一个共享的表空间里，而不是像大多数其他存储引擎那样为不同的数据表创建不同的文件。InnoDB 表空间可以由多个文件构成，还可以包括多个原始分区。实际上，InnoDB 表空间就像是一个虚拟的文件系统，它存储和管理所有 InnoDB 数据表的内容。这样一来，数据表的长度就可以超过文件系统对各个文件的最大长度的限制。你也可以把 InnoDB 存储引擎配置成会为每个数据表分别创建一个表空间的样子，此时，每个数据表在它的数据库子目录里都有一个对应的.ibd 文件。

7. Falcon存储引擎

Falcon 存储引擎是从 MySQL 6.0 开始才有的，它有以下功能。

- ❑ 支持提交和回滚操作，确保数据在事务处理过程中万无一失。还可以通过创建保存点来实现部分回滚。
- ❑ 在系统崩溃后可以自动恢复。
- ❑ 灵活的锁定级别和多版本共存，这使得 Falcon 数据表在需要同时进行检索和更新操作的复杂查询里表现出非常好的并发性能。
- ❑ 在存储时对数据行进行压缩，在检索时对数据行进行解压缩以节省空间。
- ❑ 日常管理和维护方面的开销低。

8. FEDREATED存储引擎

FEDREATED 存储引擎的用途是访问其他 MySQL 服务器管理下的数据表。换句话说，FEDREATED 数据表的内容不是存放在本地主机里的。当你创建一个 FEDREATED 数据表时，需要指定一台运行着其他服务器程序的主机，并提供那个服务器的合法账户的用户名和口令。当你打算访问 FEDREATED 数据表时，本地服务器将使用这个账户连接那台远程服务器。2.6.2 节中的第 7 小节给出了一个这样的例子。

9. NDB存储引擎

NDB 是 MySQL 的*集群*（cluster）存储引擎。在这个存储引擎工作时，MySQL 服务器的作用是帮助其他进程访问 NDB 数据表，从整个集群的高度看，其行为像是一个客户。各集群结点上的进程通过彼此通信来管理内存中的数据表。为了减少冗余，数据表在集群进程中被复制。内存存储提供了高性能，集群机制提供了高度可用性，即使个别结点发生故障，系统也不会崩溃。

NDB 存储引擎的配置和使用超出了本书的讨论范围，这里就不再多说了。有兴趣的读者请自行研读《MySQL 参考手册》。

10. 其他存储引擎

MySQL 还有几种存储引擎是前面没有提到的，如下所示。

- ARCHIVE 存储引擎对数据进行归档。它最适合用来大批量地保存那些“写了就不改”的数据行。因此，它只支持很有限的几条 SQL 语句。INSERT 和 SELECT 语句没问题，但 REPLACE 语句的行为却永远像是 INSERT 语句，而 DELETE 或 UPDATE 语句根本不能用。为了节省空间，在存储时会对数据行进行压缩，在检索时再对它们进行解压缩。在 MySQL 5.1.6 之前的版本里，ARCHIVE 存储引擎根本不支持索引；即使是在 MySQL 5.1.6 版本里，每个 ARCHIVE 数据表最多也只能有一个带索引的 AUTO_INCREMENT 数据列；其他数据列还是不能带索引。
- BLACKHOLE 存储引擎创建的数据表有这样的行为特点：写操作其实是删除数据，而读操作是返回空白记录。
- CSV 存储引擎在存储数据时以逗号作为数据项之间的分隔符。它会在数据库子目录里为每个数据表创建一个.CSV 文件。这是一种普通文本文件，每个数据行占用一个文本行。CSV 存储引擎不支持索引。
- EXAMPLE 存储引擎是用来演示如何编写存储引擎的最小化样板。它的存在价值在于让开发人员通过查看其源代码去学习怎样才能正确地把存储引擎加载到服务器里。

11. 存储引擎的可移植性

从某种意义上讲，任何一个 MySQL 服务器所管理的任何数据表都可以移植到另一台服务器上去：先用 mysqldump 工具把它备份出来，然后把备份文本文件放到另一台服务器主机，并通过加载备份文件的办法重新创建该数据表。可移植性概念还有另一层含义，即*二进制可移植性*（binary portablity），指的是你可以直接把代表某个数据表的硬盘文件复制到另一台机器，并把它们安装到数据子目录下的相应地点，然后那台机器上的 MySQL 服务器就可以使用该数据表了。

数据表具备二进制可移植性的一项基本条件是源服务器和目标服务器的有关功能必须兼容。比如说，目标服务器必须支持用来管理数据表的存储引擎。如果目标服务器上没有适用的存储引擎，它将无法访问你在源服务器上使用那种存储引擎创建的数据表。

有些存储引擎创建的数据表具备良好的二进制可移植性，有些存储引擎则不然。下面是对各个存储引擎的二进制可移植性的总结。

- MyISAM 和 InnoDB 数据表的存储格式与机器无关，它们具备二进制可移植性——前提条件是你的处理器使用的是二进制补码整数算法和 IEEE 浮点格式。一般来说，只要使用的机器不是古怪少见的品牌，这两个前提条件就不会构成真正的问题。在实践中，只要使用的不是为某种专用设备而定制的嵌入式服务器，就不太容易因为硬件因素而遇到可移植性问题，这是因为专用设备所使用的处理器往往会有一些非标准的特性。
- MERGE 数据表的可移植性取决于其成员 MyISAM 数据表，只要那些 MyISAM 数据表是可移植的，MERGE 数据表就是可移植的。
- MEMORY 数据表不具备二进制可移植性，因为它们的内容都存储在内存里而不是硬盘上。
- CSV 数据表是二进制可移植的，因为.CSV 文件在本质上都是普通的文本文件。
- BLACKHOLE 数据表是二进制可移植的，因为它们根本不包含任何内容。
- 对于 FEDERATED 存储引擎，可移植性概念与之无关，因为 FEDERATED 数据表的内容都存储在另一个服务器上。
- Falcon 日志和表空间文件的存储格式与机器有关，它们只在两台机器的硬件特性完全相同时才是二进制可移植的。比如说，不能把一台低字节优先的机器里的 Falcon 文件移植到一台高字节优先的机器。

我们刚刚提到，在两台机器之间移植 MyISAM 和 InnoDB 数据表时，在二进制可移植性方面需要满足的前提条件是：将被移植的数据表不包含任何浮点数据列，或者两台机器使用的浮点数存储格式是一样的。这里所说的“浮点数”指的是 FLOAT 和 DOUBLE 类型，不包括 DECIMAL 类型。DECIMAL 数据列里的数据值的小数点位置是固定的，这种存储格式是可移植的。

对 InnoDB 存储引擎而言，二进制可移植性还有另外一个条件：数据库和数据表的名字应该由小写字母构成。InnoDB 存储引擎在其数据字典里把这些名字统一保存为小写字母格式，但.frm 文件名里的字母却与你在 CREATE TABLE 语句里使用的大小写情况完全一样。如果当初在创建数据库或数据表时使用的是大写字母，而现在你想把它们移植到一个对文件名区分大小写的平台上去，就有可能因为字母的大小写情况不匹配而遇到问题。

对 InnoDB 存储引擎而言，必须把所有 InnoDB 数据表作为一个整体来评估二进制可移植性，而不是只考虑某一个或某几个特定的 InnoDB 数据表。在默认的情况下，InnoDB 存储引擎会把由它负责管理的所有数据表集中存储在一个共享的表空间里，而不是为各数据表分别创建一个文件。因此，不应该只考虑某一个或某几个 InnoDB 数据表是不是可移植的，必须考虑 InnoDB 表空间文件是不是可移植的。这意味着关于浮点数的可移植性规则必须针对所有包含浮点数的 InnoDB 数据表进行考虑。要知道，即使把 InnoDB 存储引擎配置成把每个数据表分别存储在一个表空间里的样子，它也会把其数据字典里的数据项集中保存在那个共享的表空间里。

无论存储引擎的可移植性怎样，都要注意：不要在服务器关闭之后把数据表或表空间文件复制到另一台机器上，除非你能确定服务器的关机操作没有任何问题。这是因为，如果服务器意外关闭，在它关闭后得到的副本将无法确保数据的完整性不受影响。你打算复制的数据表有可能需要修复，也有可能还有一些事务信息仍保存在存储引擎的日志文件里，必须等它们被提交或回滚之后才能更新数据表。

在某些情况下，可以让一个运行中的服务器在我们复制数据表文件时不要使用有关的数据表。但一般而言，只要服务器正在运行并在刷新数据表，或者还有一些改动仍缓存在内存里，硬盘上的数据表内容就仍有可能发生变化，而数据表副本就有可能没有任何实用价值。如果想知道需要满足哪些条件才能在无须关闭服务器的前提下复制数据表，请参阅 14.2 节。

2.6.2 创建数据表

创建数据表需要用到 CREATE TABLE 语句。这条语句的完整语法相当复杂，因为它的可选子句实在是太多了。还好，在实际工作中用到的绝大多数 CREATE TABLE 语句都比较简单。比如说，第 1 章用到的绝大多数 CREATE TABLE 语句都算不上复杂。只要从最基本的语法形式开始循序渐进，就应该不会遇到太大的麻烦。

最简单的 CREATE TABLE 语句只需你给出一个数据表的名字和其中数据列的名单。比如说：

```
CREATE TABLE mytbl
(
  name   CHAR(20),
  birth  DATE NOT NULL,
  weight INT,
  sex    ENUM('F','M')
);
```

在创建数据表时，除了各数据列的定义，还可以指定如何为它创建索引。另一种做法是先创建一

个不带任何索引的数据表，过后再给它加上索引。对 MyISAM 数据表来说，如果在开始对它查询之前需要填充大量的数据，第二种策略往往更好。与先把数据加载到一个不带任何索引的 MyISAM 数据表里之后再去创建索引的做法相比，每插入一个数据行都刷新一次索引要慢得多。

我们在第 1 章已经对 CREATE TABLE 语句的基本语法进行了介绍。定义数据列的细节见第 3 章。在这一小节，我们将重点介绍 CREATE TABLE 语句的几种重要的变体，它们可以帮助你灵活地构造数据表，如下所示。

- 改变存储特性的数据表选项。
- 只在数据表不存在时才创建。
- 临时数据表，服务器会在客户会话结束时自动删除它们。
- 从另一个数据表或是从一次 SELECT 查询的结果来创建数据表。
- 使用 MERGE 数据表、分区数据表、FEDERATED 数据表。

1. 数据表选项

要想改变某个数据表的存储特性，在 CREATE TABLE 语句中的右括号之后加上一个或多个数据表选项即可。数据表选项的完整清单可以在附录 E 里对 CREATE TABLE 语句的描述里查到。

有一个数据表选项是 ENGINE = *engine_name*，它用来指定用哪种存储引擎来管理将要创建的数据表。比如说，如果想创建一个 MEMORY 或 InnoDB 数据表，就要写出如下所示的语句：

```
CREATE TABLE mytbl ( ... ) ENGINE = MEMORY;
CREATE TABLE mytbl ( ... ) ENGINE = InnoDB;
```

存储引擎的名字不区分字母的大小写。如果没有给出 ENGINE 选项，服务器将使用默认的存储引擎来创建数据表。内建的默认存储引擎是 MyISAM，但你可以通过使用--default-storage-engine 选项来启动服务器，使用另外一种默认的存储引擎。在服务器运行期间，还可以通过设置系统选项 storage-engine 改变默认的存储引擎。

在 MySQL 5.0 里，一个运行中的服务器可以同时使用的存储引擎是有限的，有几种存储引擎总是可用，另外几种就要由你来挑选和配置了。如果在一条 CREATE TABLE 语句里给出了一个服务器能够支持但在此时此刻不可用的存储引擎的名字，MySQL 将使用默认的存储引擎去创建那个数据表并生成一条警告信息。比如说，如果 ARCHIVE 存储引擎是服务器能够支持但在此时此刻不可用的，你在创建一个 ARCHIVE 数据表时就会看到如下所示的消息：

```
mysql> CREATE TABLE t (i INT) ENGINE = ARCHIVE;
Query OK, 0 rows affected, 1 warning (0.01 sec)
mysql> SHOW WARNINGS;
+---------+------+----------------------------------------+
| Level   | Code | Message                                |
+---------+------+----------------------------------------+
| Warning | 1266 | Using storage engine MyISAM for table 't' |
+---------+------+----------------------------------------+
```

如果给出的存储引擎的名字是服务器不支持的，它将报告一条出错消息。

在 MySQL 5.1 和更高的版本里，服务器使用了一种插入式的体系结构，这使得服务器可以在运行时动态地加载存储引擎。“服务器可以使用的存储引擎”现在的含义应该是“服务器当前已加载的存储引擎”。如果在创建数据表时给出了一个此时此刻尚未加载的存储引擎的名字，服务器将生成两条警告消息：

```
mysql> CREATE TABLE t (i INT) ENGINE = ARCHIVE;
Query OK, 0 rows affected, 2 warnings (0.01 sec)
mysql> SHOW WARNINGS;
+---------+------+-------------------------------------------+
| Level   | Code | Message                                   |
+---------+------+-------------------------------------------+
| Warning | 1286 | Unknown table engine 'ARCHIVE'            |
| Warning | 1266 | Using storage engine MyISAM for table 't' |
+---------+------+-------------------------------------------+
```

要想让某个数据表使用特定的存储引擎，就一定要给出相应的 ENGINE 数据表选项。默认的存储引擎可以改变，所以省略 ENGINE 选项有可能导致你预期使用的默认存储引擎与实际不一样。此外，一定要保证 CREATE TABLE 语句没有生成任何警告消息，与这条语句有关的警告消息基本上都是“某某存储引擎不可用，使用了默认的存储引擎”。

如果不想让 MySQL 在指定的存储引擎不可用时使用默认的存储引擎代替之，需要激活 NO_ENGINE_SUBSTITUTION SQL 模式。

如果想知道数据表使用的是哪一种存储引擎，发出一条 SHOW CREATE TABLE 语句并查看其输出内容里的 ENGINE 选项即可：

```
mysql> SHOW CREATE TABLE t\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `i` int(11) DEFAULT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

存储引擎还可以通过 SHOW TABLE STATUS 语句或 INFORMATION_SCHEMA.TABLES 数据表查看。

有些数据表选项只适用于特定的存储引擎。比如说，在创建 MEMORY 数据表时加上一个 MIN_ROWS=*n* 选项可能会很有用，它可以让 MEMORY 存储引擎对内存的使用情况进行优化：

```
CREATE TABLE mytbl ( ... ) ENGINE = MEMORY MIN_ROWS = 10000;
```

如果 MEMORY 引擎认为 MIN_ROWS 的值足够大，它在分配内存时就会使用大内存块以避免因为需要发出太多的内存分配调用而增加总体开销。

MAX_ROWS 和 AVG_ROW_LENGTH 选项可以帮你在某种程度上控制 MyISAM 数据表的大小。在默认的情况下，MyISAM 存储引擎在创建数据表时使用的内部数据行指针的长度允许数据表文件增长到 256 TB。如果你给出了 MAX_ROWS 和 AVG_ROW_LENGTH 选项，MyISAM 存储引擎就会根据这些信息为数据表选用一个适当的内部数据行指针长度，确保它至少能够容纳 MAX_ROWS 个数据行。

如果想改变现有的数据表的存储特性，在 ALTER TABLE 语句里写出相关的数据表选项即可。比如说，如果想把 mytbl 数据表现在使用的存储引擎改成 InnoDB，发出下面这条语句即可：

```
ALTER TABLE mytbl ENGINE = InnoDB;
```

如果想了解更多关于如何改变存储引擎的信息，请参阅 2.6.5 节。

2. 只创建原本没有的数据表

如果你只想创建原本没有的数据表，请使用 CREATE TABLE IF NOT EXIST 语句。这条语句可以让你的应用程序无须假设它需要用到的数据表是否已经存在。你的应用程序将去尝试创建那个数据表，无论它是否已经存在都不会影响应用程序的正常执行。IF NOT EXIST 短语在你打算通过 mysql

工具去执行的批处理脚本里非常有用。对于这种情况，普通的 CREATE TABLE 语句存在一个小问题：在你第一次运行批处理脚本时，它将正常地创建出那个数据表，但在第二次运行它的时候就会出错，因为那个数据表已经存在了。如果你使用了 IF NOT EXIST 短语，就不会发生这样的问题了：在第一次运行批处理脚本时，它将正常地创建出那个数据表。在第二次和以后运行这个脚本时，尝试创建那个数据表的操作将被毫无声息地忽略，不会产生任何错误，而你的批处理任务将继续往下执行，就像这次尝试取得了成功那样。

如果打算使用 IF NOT EXIST 短语，有个细节必须注意：MySQL 不会去比较 CREATE TABLE 语句里的数据表的结构与已经存在的那个数据表是否一致。即使已经存在一个名字相同、但结构不同的数据表，这条语句也不会报告出错，而你后面的操作可就会乱套了。如果不想冒这个险，可以先执行一条 DROP TABLE IF EXIST 语句，再执行一条不带 IF NOT EXIST 短语的 CREATE TABLE 语句。

3. 临时数据表

如果在数据表创建语句里加上 TEMPORARY 关键字，服务器将创建出一个临时的数据表，它在你与服务器的连接断开时自动消失：

```
CREATE TEMPORARY TABLE tbl_name ... ;
```

这么做的好处是你不必惦记还得发出一条 DROP TABLE 语句来删除那个数据表，即使你与服务器的连接意外断开了，那个数据表也不会成为“流浪汉”。比如说，假设你有一组复杂的查询命令保存在一个批处理文件里，你通过 mysql 工具运行了那个批处理文件但临时决定不等它执行完毕，你可以毫无顾虑地“杀”掉那个脚本，而服务器将删除由该脚本创建的所有临时数据表。

如果想使用某种存储引擎来创建临时数据表，给 CREATE TEMPORARY TABLE 语句加上 ENGINE 数据表选项即可。

服务器会在你的客户会话结束时自动删除一个 TEMPORARY 数据表，但它也完全可以在用完它之后显式地删除它，这可以让服务器尽快释放与之相关联的资源。如果你与服务器之间的会话还需要再持续一段时间，及时释放不再需要的资源是一种良好习惯，尤其是那些临时性的 MEMORY 数据表。

一个 TEMOPARY 数据表只对创建该数据表的客户是可见的。因为每个客户只能看到它自己创建的数据表，所以不同的客户可以各自创建一个名字相同的 TEMPORARY 数据表而不会发生冲突。

一个 TEMPORARY 数据表的名字允许与一个现有的永久性数据表相同。这不是一个错误，那个现有的永久性数据表也不会因此遭到损坏。只是创建这个 TEMPORARY 数据表的客户在此表存在期间将不再能够看见（也就是无法访问）那个永久性数据表而已。比如说，如果你在 sampdb 数据库中创建了一个名为 member 的 TEMPORARY 数据表，原有的 member 数据表将被隐藏起来，对 member 数据表的访问将作用于新建的 TEMPORARY 数据表。如果你发出一条 DROP TABLE member 语句，被删除的将是那个 TEMPORARY 数据表，原有的 member 数据表将重现在你眼前。如果你在没有删除那个 TEMPORARY 数据表的情况下断开了与服务器的连接，服务器将自动地替你删除它。等你下次连接的时候，原有的 member 数据表就又是可见的了。（如果把一个 TEMPORARY 数据表重新命名为另外一个名字，原有的数据表也会变成可见的。）

这种“隐姓埋名”机制只给你一次机会，因为你无法创建两个同名的 TEMPORARY 数据表。

在考虑要不要使用 TEMPORARY 数据表时，请注意以下几个因素。

- 如果客户程序会在与服务器的连接意外断开时自动重建连接，上次创建的 TEMPORARY 数据表在你重新连接上服务器时将不复存在。如果使用 TEMPORARY 数据表的目的是为了“隐藏”一

个与之同名的永久性数据表，那个永久性的数据表现在就会变成可用的，而这就带来了一定的风险。比如说，如果连接意外断开后立刻得到重建但你没有察觉，你发出的 DROP TABLE 语句将会导致那个永久性的数据表被删除。如果想避免这种问题，就应该使用 DROP TEMPORARY TABLE 语句来代替之。

- 因为 TEMPORARY 数据表只对创建它们的连接是可见的，所以如果使用了某种连接池机制，它们的用处就没有多大了，因为连接池机制不能保证你发出的每一条语句使用的都是同一条连接。
- 如果使用了连接池或永久性连接，你与 MySQL 服务器之间的连接在应用程序结束时就不一定会被关闭。那些机制可能会让连接保持打开状态供其他客户使用，而这意味着你创建的 TEMPORARY 数据表不一定会在你的应用程序结束时自动消失。

4. 从其他数据表或查询结果创建数据表

在某些场合，为某个数据表创建一份副本很有必要。比如说，你有一个数据文件，你想用 LOAD DATA 语句把它添加到某个数据表里，但你对用来给出数据格式的选项没有把握。万一那些选项设置得不正确，你就会把一些乱七八糟的数据行插入到原始数据表里。如果你有一份原始数据表的空白副本，你就可以放心大胆地通过尝试各种 LOAD DATA 选项的办法来确定那个数据文件使用的数据列和数据行分隔符是什么。等你认为来自数据文件的输入行得到了正确的解析之后，只需再次运行 LOAD DATA 语句并在该语句里给出那个原版数据表的名字，就可以把你的数据文件加载到那个原版数据表里去了。

在另外一些场合，把查询结果保存到一个数据表里要比让它们从显示器的屏幕一闪而过更符合我们的愿望。保存起来的查询结果使我们无需再次运行原始查询命令就可以使用它们，尤其是在需要对它们做进一步分析的时候。

MySQL 提供了两条语句来帮助我们从其他的数据表或是从查询结果创建新的数据表。这两条语句各有各的优点和缺点，如下所示。

- CREATE TABLE...LIKE 语句将创建一个新数据表作为原始数据表的一份空白副本。它将把原始数据表的结构丝毫不差地复制过来，不仅各数据列的所有属性都会得到保留，就连索引结构也会复制得一模一样。不过，因为新数据表的内容是一片空白，所以如果想填充它，就需要再使用一条语句（如 INSERT INTO...SELECT）。请注意，CREATE TABLE...LIKE 语句不能从原始数据表的数据列的一个子集创建出一个新数据表，它也不能使用除原始数据表以外的任何其他数据表里的数据列。
- CREATE TABLE...SELECT 语句可以从任意一条 SELECT 语句的查询结果创建新数据表。在默认的情况下，这条语句不会复制所有的数据列属性，如 AUTO_INCREMENT 等。通过选取数据到其中而创建新数据表也不会自动复制原始数据表里的所有索引，因为结果集本身不带索引。从另一方面讲，CREATE TABLE...SELECT 语句只需一条语句就可以完成创建和填充新数据表两项任务。它还可以用原始数据表的一个子集创建一个新数据表，并包括来自其他数据表的数据列或作为表达式结果而被创建出来的数据列。

使用 CREATE TABLE...LIKE 语句为一个现有的数据表创建一份空白副本的基本语法如下所示：

```
CREATE TABLE new_tbl_name LIKE tbl_name;
```

为数据表创建一份空白副本，再从原始数据表填充它，需要先使用一条 CREATE TABLE...LIKE 语句，再使用一条 INSERT INTO...SELECT 语句：

```
CREATE TABLE new_tbl_name LIKE tbl_name;
```

```
INSERT INTO new_tbl_name SELECT * FROM tbl_name;
```

如果想创建一个数据表作为它本身的一个临时副本，需要加上 TEMPORARY 关键字：

```
CREATE TEMPORARY TABLE new_tbl_name LIKE tbl_name;
INSERT INTO new_tbl_name SELECT * FROM tbl_name;
```

创建一个与原始数据表同名的 TEMPORARY 数据表，这在你打算使用一些语句去修改该数据表的内容、但又不想改变原始数据表的内容时会很有用。比如说，如果在事先编写好的脚本里使用的是原始数据表的名字，你无需改写脚本使用另外一个数据表，只要在脚本的开头加上一条 CREATE TEMPORARY TABLE 语句和一条 INSERT 语句就可以了。你的脚本将创建一份临时副本并在该份副本上进行各种操作，当脚本结束时，服务器会自动删除它。（不过，千万要注意上一小节提到的自动重建连接的问题。）

如果只想把原始数据表里的一部分数据行插入到新数据表里，可以增加一条 WHERE 子句来选取有关的数据行。下面的语句将创建一个名为 student_f 的新数据表，它只包含 student 数据表里的女学生的数据行：

```
CREATE TABLE student_f LIKE student;
INSERT INTO student_f SELECT * FROM student WHERE sex = 'f';
```

如果不关心新数据表是否保留了原始数据表里的数据列的精确定义，CREATE TABLE...SELECT 语句有时要比 CREATE TABLE...LIKE 语句更容易使用，因为前者只需一条语句就可以创建并填充新数据表：

```
CREATE TABLE student_f SELECT * FROM student WHERE sex = 'f';
```

用 CREATE TABLE...SELECT 语句创建出来的新数据表并非只能包含来自某一个数据表的数据列。你随时都可以在必要时用它快速创建一个新数据表来容纳任何一次 SELECT 查询的结果。这让我们可以非常简便快速地创建一个新数据表并让该数据表的内容都成为我们想要的数据（供后面的语句使用）。不过，如果不小心，新数据表可能会包含一些奇怪的数据列名字。当你通过选取数据到其中而创建数据表时，数据列的名字来自你正在选取的数据列。如果某个数据列是一个表达式的计算结果，该数据列的名字将是表达式的文本，而如此创建出来的数据表将包含一个不同寻常的数据列名字，如下所示：

```
mysql> CREATE TABLE mytbl SELECT PI() * 2;
mysql> SELECT * FROM mytbl;
+----------+
| PI() * 2 |
+----------+
| 6.283185 |
+----------+
```

这显然不理想，因为这样的数据列名字只能以一个用引号括起来的标识符的形式被直接引用：

```
mysql> SELECT `PI() * 2` FROM mytbl;
+----------+
| PI() * 2 |
+----------+
| 6.283185 |
+----------+
```

要想避免这个问题并提供一个便于使用的数据列名字，可以使用一个别名：

```
mysql> DROP TABLE mytbl;
mysql> CREATE TABLE mytbl SELECT PI() * 2 AS mycol;
mysql> SELECT mycol FROM mytbl;
+----------+
| mycol    |
+----------+
| 6.283185 |
+----------+
```

使用别名会带来一个小问题：如果你还从另一个数据表选取了一个同名的数据列，就会发生冲突。比如说，假设数据表 t1 和 t2 都有一个数据列 c，而你想从这两个数据表的所有数据行的组合里创建一个数据表。下面的语句将失败，因为它试图创建的数据表里有冲突——两个数据列的名字都是 c：

```
mysql> CREATE TABLE t3 SELECT * FROM t1 INNER JOIN t2;
ERROR 1060 (42S21): Duplicate column name 'c'
```

这个问题并不难解决，通过提供必要的别名让两个数据列在新数据表里各有一个独一无二的名字就行了：

```
mysql> CREATE TABLE t3 SELECT t1.c, t2.c AS c2
    -> FROM t1 INNER JOIN t2;
```

正如刚才提到的那样，CREATE TABLE...SELECT 语句的缺点之一是它不会把原始数据的所有特征全部复制到新数据表的结构里去。比如说，通过选取数据到其中来创建一个数据表不会把原始数据表里的索引复制过去，而且还可能失去数据列属性。可以保留下来的属性包括数据列是 NULL 还是 NOT NULL、字符集和排序方式、默认值和数据列注释。

在某些场合，你可以通过在语句的 SELECT 部分使用 CAST()函数的办法在新数据表里强制使用特定的属性。下面的 CREATE TABLE...SELECT 语句将强制由 SELECT 子句所生成的数据列被视为 INT UNSIGNED、TIME 和 DECIMAL(10,5)来对待，就像用 DESCRIBE 语句进行验证时看到的那样：

```
mysql> CREATE TABLE mytbl SELECT
    -> CAST(1 AS UNSIGNED) AS i,
    -> CAST(CURTIME() AS TIME) AS t,
    -> CAST(PI() AS DECIMAL(10,5)) AS d;
mysql> DESCRIBE mytbl;
+-------+-----------------+------+-----+---------+-------+
| Field | Type            | Null | Key | Default | Extra |
+-------+-----------------+------+-----+---------+-------+
| i     | int(1) unsigned | NO   |     | 0       |       |
| t     | time            | YES  |     | NULL    |       |
| d     | decimal(10,5)   | NO   |     | 0.00000 |       |
+-------+-----------------+------+-----+---------+-------+
```

允许的投射类型是 BINARY(二进制串)、CHAR、DATE、DATETIME、TIME、SIGNED、SIGNED INTEGER、UNSIGNED、UNSIGNED INTEGER 和 DECIMAL。

还可以在 CRAETE TABLE 部分提供明确的数据列定义，然后在 SELECT 部分使用那些定义去检索数据列。两个部分中的数据列按名字匹配，所以在 SELECT 部分往往需要提供一些必要的别名才能让它们正确地得到匹配：

```
mysql> CREATE TABLE mytbl (i INT UNSIGNED, t TIME, d DECIMAL(10,5))
    -> SELECT
    -> 1 AS i,
```

```
    -> CAST(CURTIME() AS TIME) AS t,
    -> CAST(PI() AS DECIMAL(10,5)) AS d;
mysql> DESCRIBE mytbl;
+-------+------------------+------+-----+---------+-------+
| Field | Type             | Null | Key | Default | Extra |
+-------+------------------+------+-----+---------+-------+
| i     | int(10) unsigned | YES  |     | NULL    |       |
| t     | time             | YES  |     | NULL    |       |
| d     | decimal(10,5)    | YES  |     | NULL    |       |
+-------+------------------+------+-----+---------+-------+
```

提供明确的数据列定义的技巧使你能够创建具有特定精度和取值范围的数值数据列、与结果集中最长的值的宽度不同的字符数据列,等等。此外,请注意在这个例子里有几个数据列的Null和Default属性与前面几个例子里的情况不一样。如有必要，你可以在 CREATE TABLE 部分为这些属性提供明确的定义。

5. 使用MERGE数据表

MERGE 存储引擎把一组 MyISAM 数据表当做一个逻辑单元来对待，让我们可以同时对它们进行查询。正如在前面 2.6.1 节所描述的，构成一个 MERGE 数据表的各成员 MyISAM 数据表必须具有完全一样的结构。每一个成员数据表里的数据列必须按照同样的顺序定义同样的名字和类型，索引也必须按照同样的顺序和同样的方式定义。

假设你有几个日志数据表，它们的内容是分别是这几年来每一年里的日志记录项，它们的定义都是下面这样，其中 *CC* 代表世纪，*YY* 代表年份：

```
CREATE TABLE log_CCYY
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
) ENGINE = MyISAM;
```

假设日志数据表的当前集合包括 log_2004、log_2005、log_2006、log_2007 和 log_2008，而你可以创建一个如下所示的 MERGE 数据表把它们归拢为一个逻辑单元：

```
CREATE TABLE log_merge
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
) ENGINE = MERGE UNION = (log_2004, log_2005, log_2006, log_2007, log_2008);
```

ENGINE 选项的值必须是 MERGE，UNION 选项列出了将被收录在这个 MERGE 数据表里的各有关数据表。把这个 MERGE 数据表创建出来之后，你可以像对待任何其他的数据表那样查询它，只是每次查询都将同时作用于构成它的每一个成员数据表。下面这个查询可以让我们知道上述几个日志数据表里的数据行的总数：

```
SELECT COUNT(*) FROM log_merge;
```

下面这个查询用来确定在这几年里每年各有多少条日志记录项：

```
SELECT YEAR(dt) AS y, COUNT(*) AS entries FROM log_merge GROUP BY y;
```

除了便于同时引用多个数据表而无须发出多条查询命令，MERGE 数据表还提供了以下一些便利。

- MERGE 数据表可以用来创建一个尺寸超过各个 MyISAM 数据表所允许的最大长度的逻辑单元。
- 你可以把经过压缩的数据表包括到 MERGE 数据表里。比如说，在某一年结束之后，你应该不会再往相应的日志文件里添加任何记录了，所以你可以用 myisampack 工具压缩它以节省空间，而 MERGE 数据表仍可以像以前那样使用。

MERGE 数据表也支持 `DELETE` 和 `UPDATE` 操作。`INSERT` 操作比较麻烦，因为 MySQL 需要知道应该把新数据行插入到哪一个成员数据表里去。在 MERGE 数据表的定义里可以包括一个 `INSERT_METHOD` 选项，这个选项的可取值是 `NO`、`FIRST` 和 `LAST`，它们的含义依次是 `INSERT` 操作是被禁止的、新数据行将被插入到在 `UNION` 选项里列出的第一个数据表或最后一个数据表。比如说，如下所示的定义将使对 `log_merge` 数据表的 `INSERT` 操作被当做对 `log_2008` 数据表——它是 UNION 选项所列出的最后一个数据表——的一个 `INSERT` 操作：

```
CREATE TABLE log_merge
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
) ENGINE = MERGE UNION = (log_2004, log_2005, log_2006, log_2007, log_2008)
INSERT_METHOD = LAST;
```

创建一个新的成员数据表 `log_2009` 并让它有着与其他 `log_`*`CCYY`* 数据表同样的结构，然后修改 `log_merge` 数据表的定义把 `log_2009` 包括进来即可：

```
log_2009:
CREATE TABLE log_2009 LIKE log_2008;
ALTER TABLE log_merge
UNION = (log_2004, log_2005, log_2006, log_2007, log_2008, log_2009);
```

6. 使用分区数据表

MySQL 5.1 及更高版本支持*分区数据表*（partitioned table）。分区在概念上与 MERGE 存储引擎很相似：它们都可以用来访问被分别存储在不同地点的多个数据表的内容。这两者之间的区别是：每个分区数据表都是一个货真价实的数据表，而不是一个用来列出各成员数据表的逻辑构造。此外，分区数据表可以使用 MyISAM 以外的存储引擎，而 MERGE 数据表只能用 MyISAM 数据表来构成。

通过对数据表的存储进行划分，分区数据表具有以下几个优点。

- 数据表的存储可以分布在多个设备上，这意味着我们可以通过建立某种 I/O 并行机制缩短访问时间。
- 优化器可以把检索操作限定在某个特定的分区或是同时搜索多个分区。

在创建一个分区数据表的时候，先像往常一样在 `CREATE TABLE` 语句里给出数据列和索引的清单，然后用一条 `PARTITION BY` 子句定义一个用来把数据行分配到各个分区的分区函数，再写出其他必须的分区选项即可。分区函数的作用类似于我们在创建 MERGE 数据表时使用的 `INSERT_METHOD` 选项，只是它更通用：它可以把新数据行分布到所有的分区，而 `INSERT_METHOD` 选项只能把所有的新数据行插入到同一个数据表。

分区函数把新数据行分配到不同分区的依据可以是取值范围、值的列表或散列值，如下所示。

- 根据取值范围进行分区。数据行所包含的值可以划分为一系列互不冲突的区间，比如日期、收入水平、重量等。

- 根据列表进行分区。每个分区分别对应一个明确的值的列表，比如邮政编码表、电话号码区号、身份证号码中的地区编号等。
- 根据散列值进行分区。根据数据行的键字计算出一个散列值，再根据这个散列值把数据行分布到各分区。你可以自行提供一个散列函数，也可以列出一组数据列让MySQL使用一个内建的散列函数去计算那些数据列的散列值。

分区函数必须具备这样一种确定性：同样的输入永远会把数据行分配到同一个分区。按照这一要求，诸如 RAND()和 NOW()之类的函数显然不适合用做分区函数。

作为一个简单的例子，让我们一起来为第 5 小节里的 MERGE 数据表创建一个分区数据表版本。那个名为 log_merge 的 MERGE 数据表由多个成员数据表构成，它们的内容分别是 2004 至 2008 年期间每一年里的日志记录项。相应的分区数据表将是一个被划分为多个分区的数据表。因为在构成日志记录项的数据里肯定会包含一个日期值，所以根据这个日期值的取值范围进行分区是最顺理成章的办法。我决定根据年份（也就是日期值里的“年”部分）来把数据行分配到一个给定的分区：

```
CREATE TABLE log_partition
(
  dt    DATETIME NOT NULL,
  info  VARCHAR(100) NOT NULL,
  INDEX (dt)
)
PARTITION BY RANGE(YEAR(dt))
(
  PARTITION p0 VALUES LESS THAN (2005),
  PARTITION p1 VALUES LESS THAN (2006),
  PARTITION p2 VALUES LESS THAN (2007),
  PARTITION p3 VALUES LESS THAN (2008),
  PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

根据上面的定义，2008 年和以后的日志记录项将被分配到 MAXVALUE 分区。2009 年时可以对这个分区再进行划分，把 2008 年的日志记录项转移到它们自己的一个分区里，把 2009 年和以后的日志记录项仍保留在 MAXVALUE 分区里，如下所示：

```
ALTER TABLE log_partition REORGANIZE PARTITION p4
INTO (
  PARTITION p4 VALUES LESS THAN (2009),
  PARTITION p5 VALUES LESS THAN MAXVALUE
);
```

在默认的情况下，分区被保存在分区数据表所属于的数据库的子目录里。若想将存储分配到其他位置（如另一个物理设备），需要使用 DATA_DIRECTORY 和 INDEX_DIRECTORY 分区选项。如果想了解关于这两个以及其他分区选项的语法的更多信息，请参阅附录 E 中对 CREATE TABLE 语句的描述。

7. 使用FEDERATED数据表

FEDERATED 存储引擎可以让你访问在其他主机上由另一个 MySQL 服务器实际管理的数据表。

假设你的本地服务器上有一个名为 sampdb 的数据库，在网址是 corn.snake.net 的主机上还有一个 MySQL 服务器也管理着一个同名的数据库，而你有一个账户可以访问那个远程服务器。这样一来，你就可以使用那个账户通过 FEDERATED 存储引擎在本地主机上使用位于远程主机的 sampdb 数据库。对于每一个你想如此访问的数据表，你必须创建一个与远程数据表有着同样数据列的 FEDE-

RATED 数据表，并给出相应的连接字符串让本地服务器知道如何连接远程服务器。

假设远程服务器上的 student 数据表有着如下所示的定义：

```
CREATE TABLE student
(
  name       VARCHAR(20) NOT NULL,
  sex        ENUM('F','M') NOT NULL,
  student_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (student_id)
) ENGINE = InnoDB;
```

在本地服务器上创建一个相应的 FEDERATED 数据表必须使用同样的定义，还必须把 ENGINE 选项设置为 FEDERATED，再通过 CONNECTION 连接选项给出建立连接所必需的信息。（MySQL 5.0.13 之前的版本需要用 COMMENT 选项代替 CONNECTION 选项。）如下所示的定义将创建一个名为 federated_student 的数据表，它用来访问位于 corn.snake.net 的主机上的 student 数据表：

```
CREATE TABLE federated_student
(
  name       VARCHAR(20) NOT NULL,
  sex        ENUM('F','M') NOT NULL,
  student_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (student_id)
) ENGINE = FEDERATED
CONNECTION = 'mysql://sampadm:secret@corn.snake.net/sampdb/student';
```

从 CONNECTION 选项所给出的连接字符串可以看到，用来访问远程服务器的 MySQL 账户的用户名和口令是 sampdb 和 secret。连接字符串的基本语法如下所示，方括号里是可选的信息。

```
mysql://user_name[:password]@host_name[:port_num]/db_name/tbl_name
```

把 federated_student 数据表创建出来之后，你就可以通过查询它而访问远程的 student 数据表了。你还可以通过对 federated_student 数据表进行 INSERT、UPDATE 和 DELETE 操作的办法去改变那个远程 student 数据表的内容。

这里有一个值得注意的问题：整个 CONNECTION 字符串（包括用户名和口令）对可以使用 SHOW CREATE TABLE 或类似语句去查看你的 FEDERATED 数据表的任何人来说都是可见的。从 MySQL 5.1.15 版开始，你可以避免这个问题：提前用 CREATE SERVER 语句创建一个存储服务器定义（这需要 SUPER 权限），然后在 CONNECTION 选项里写出该服务器的名字即可。下面这条语句定义了一个名为 corn_sampdb-server 的存储服务器：

```
CREATE SERVER corn_sampdb_server
FOREIGN DATA WRAPPER mysql
OPTIONS (
  USER 'sampadm',
  PASSWORD 'secret',
  HOST 'corn.snake.net',
  DATABASE 'sampdb'
);
```

MySQL 服务器将把这个定义保存为 mysql 数据库中的 servers 数据表的一个数据行。如果你想创建一个引用这个服务器定义的数据表，在如下所示的语句里通过 CONNECTION 选项给出远程服务器的名字即可：

```
CREATE TABLE federated_student2
(
  name       VARCHAR(20) NOT NULL,
  sex        ENUM('F','M') NOT NULL,
  student_id INT UNSIGNED NOT NULL AUTO_INCREMENT,
  PRIMARY KEY (student_id)
) ENGINE = FEDERATED
CONNECTION = 'corn_sampdb_server/student';
```

与把连接参数直接写在 CONNECTION 选项里的做法相比，使用服务器定义可以提高安全性，因为这种定义只有那些有权访问 mysql 数据库的用户才能查看到。此外，服务器定义还可以简化数据表的创建工作，共享着同样的连接参数的多个 FEDERATED 数据表可以使用同一个定义。

2.6.3 删除数据表

删除数据表要比创建它容易得多，因为你不需要给出任何关于其内容的格式的信息。你只需给出它的名字即可：

```
DROP TABLE tbl_name;
```

MySQL 为 DROP TABLE 语句提供了几种很有用的变体。如果需要删除多个数据表，把它们依次列在同一条语句里即可：

```
DROP TABLE tbl_name1, tbl_name2, ... ;
```

如果你不能肯定某个数据表是否已经存在，但如果它存在的话你就要删除它，请在语句里加上 IF EXIST 子句：

```
DROP TABLE IF EXISTS tbl_name;
```

IF EXIST 子句的作用是在给定的数据表不存在时不让这条语句报告错误。（但每出现一次这样的情况，服务器就会生成一条警告消息，你可以用 SHOW WARNINGS 语句去查看它们。）

IF EXIST 子句非常适合用在你将通过 mysql 客户工具去运行的脚本里。在默认的情况下，mysql 工具会在有错误发生时退出运行，而删除一个并不存在的数据表是一个错误。比如说，假设你有一个初始化脚本，它负责创建一些数据表供其他脚本做进一步处理。在这种场合里，你应该确保初始化脚本在开始运行时有一个干净的环境。如果你在这个脚本的开头使用的是普通的 DROP TABLE 语句，它在第一次运行时就会失败，因为它想要删除的数据表还都没有被创建出来。如果你使用了 IF EXIST 子句，就不会出问题了。如果数据表已经存在，它们将被删除；如果它们不存在，也不会发生错误，脚本将继续执行。

如果你只想删除临时数据表，加上 TEMPORARY 关键字即可：

```
DROP TEMPORARY TABLE tbl_name;
```

2.6.4 为数据表编制索引

索引是加快对数据表内容的访问速度的基本手段，尤其是在涉及多个数据表的关联查询里。这是一个非常重要的主题，我们在后面用了整整一章的篇幅来讨论为什么要使用索引，它们是如何工作的，以及怎样才能最大限度地利用索引去优化查询（参见第 5 章）。本小节的重点是介绍各种适用于不同数据表类型的索引的特性以及创建和删除索引的语法。

1. 存储引擎的索引特性

MySQL 提供了多种灵活的索引创建办法，如下所示。

- 你可以为单个数据列编制索引，也可以为多个数据列构造复合索引。
- 索引可以只包含独一无二的值，也可以包含重复的值。
- 你可以为同一个数据表创建多个索引并分别利用它们来优化基于不同数据列的查询。
- 对于 ENUM 和 SET 以外的字符串数据类型，可以只为数据列的一个前缀创建索引，也就是为对最左边的 n 个字符（对二进制字符串类型来说就是最左边的 n 个字节）创建索引。（对于 BLOB 和 TEXT 数据列，你只有在指定了前缀长度的情况下才能创建一个索引。）如果数据列在前缀长度范围内具有足够的独一无二性，查询性能通常不会受到影响，而是会得到改善：为数据列前缀而不是整个数据列编索引可以让索引本身更小并加快访问速度。

并非所有的存储引擎都能提供全部的索引功能。表 2-3 对 MySQL 的几种存储引擎的索引特性进行了汇总。这个表没有包括 MERGE 存储引擎，因为 MERGE 数据表是从一组 MyISAM 数据表创建出来的，它们有相似的索引特性。它也没有包括 ARCHIVE、BLACKHOLE、CSV 或 EXAMPLE 引擎，它们或者根本不支持索引，或者支持很有限。

表2-3　存储引擎的索引特性

索引特性	MyISAM	MEMOR	InnoDB	Falcon
是否允许使用NULL值	是	是	是	是
每个索引最多支持多少个数据列	16	16	16	16
每个数据表最多可以有多少个索引	64	64	64	64
索引项的最大长度（字节）	1000	1024/3072	1024/3072	1100
能否为数据列前缀创建索引	能	能	能	能
数据列前缀的最大长度（字节）	1000	1024/3072	767	1100
是否支持BLOB/TEXT索引	是	否	是	否
是否支持FULLETXT索引	是	否	否	否
是否支持SPATIAL索引	是	否	否	否
是否支持HASH索引	否	是	否	否

对于 MEMORY 和 InnoDB 存储引擎，索引项的最大长度在 MySQL 5.0.17/5.1.4 之前的版本里是 1024 个字节，在 5.0.17/5.1.4 及更高的版本里是 3 072 个字节。MEMORY 存储引擎的索引前缀的最大长度也是如此。

不同的存储引擎具备不同的索引特性，其中有这样一层含义：如果你要求某个索引必须具备某种特性，你将无法使用特定类型的数据表。比如说，如果你想使用 FULLTEXT 或 SAPTIAL 索引，就必须使用 MyISAM 数据表；如果你想对一个 TEXT 数据列编索引，就必须使用 MyISAM 或 InnoDB 数据表。

如果你想让一个现有的数据表改用另外一种有着你更需要的索引特性的存储引擎，可以使用 ALTER TABLE 语句来改变存储引擎。假设你正在使用一个 MyISAM 数据表但需要 InnoDB 或 Falcon 存储引擎提供的事务处理能力，你将需要使用如下所示的语句对数据表进行转换：

```
ALTER TABLE tbl_name ENGINE = InnoDB;
ALTER TABLE tbl_name ENGINE = Falcon;
```

2. 创建索引

MySQL 可以创建好几种索引，如下所示。

- 唯一索引。这种索引不允许索引项本身出现重复的值。对只涉及一个数据列的索引来说，这意味着该数据列不能包含重复的值。对涉及多个数据列的索引（复合索引）来说，这意味着那几个数据列的值的组合在整个数据表的范围内不能出现重复。
- 普通（非唯一）索引。这种索引的优点（从另一方面看是缺点）是允许索引值出现重复。
- FULLTEXT 索引。用来进行全文检索。这种索引只适用于 MyISAM 数据表。如果你想了解更多信息，请参阅 2.15 节。
- SPATIAL 索引。这种索引只适用于 MyISAM 数据表和空间（spatial）数据类型，对这种数据类型的描述见第 3 章。（对于其他支持空间数据类型的存储引擎，你可以创建非 SPATIAL 索引。）
- HASH 索引。这是 MEMORY 数据表的默认索引类型，但你可以改用 BTREE 索引来代替这个默认索引。

你可以在使用 CREATE TABLE 语句创建新数据表时创建索引。这方面的例子参见 1.4.6 节。用 ALTER TABLE 或 CREATE INDEX 语句可以给现有数据表添加索引。（MySQL 会在其内部把 CREATE INDEX 语句映射为 ALTER TABLE 操作。）

ALTER TABLE 语句比 CREATE INDEX 语句更灵活多能，因为它可以用来创建 MySQL 所能支持的任何一种索引。比如说：

```
ALTER TABLE tbl_name ADD INDEX index_name (index_columns);
ALTER TABLE tbl_name ADD UNIQUE index_name (index_columns);
ALTER TABLE tbl_name ADD PRIMARY KEY (index_columns);
ALTER TABLE tbl_name ADD FULLTEXT index_name (index_columns);
ALTER TABLE tbl_name ADD SPATIAL index_name (index_columns);
```

tbl_name 是你想添加索引的数据表的名字，*index_columns* 是你想加索引的一个或多个数据列。如果索引由多个数据列构成，要用逗号把它们的名字隔开。索引本身的名字 *index_name* 是可选的，如果你没有给出，MySQL 将根据第一个带索引的数据列给它挑选一个名字。

如果某个索引是一个 PRIMARY KEY 或 SPATIAL 索引，带索引的数据列必须具备 NOT NULL 属性。其他用途的索引都允许包含 NULL 值。

只需用逗号把它们彼此隔开，你就可以在同一条 ALTER TABLE 语句里包括多个对数据表的改动。这意味着你可以同时创建多个索引，这比使用多条 ALTER TABLE 语句逐个地添加索引的办法要快很多。

如果想限制某个索引只包含独一无二的值，可以把该索引创建为一个 PRIMARY KEY 或一个 UNIQUE 索引。这两种索引很相似，但有两点区别，如下所示。

- 每个数据表只能有一个 PRIMARY KEY。（这是因为 PRIMARY KEY 的名字总是 PRIMARY，而同一个数据表不允许有两个同名的索引。）
- PRIMARY KEY 不可以包含 NULL 值，而 UNIQUE 索引可以。而且，如果你允许某个 UNIQUE 索引包含 NULL 值，那它将可以包含多个 NULL 值。这是因为 MySQL 无法判断两个 NULL 值是否代表同样的东西，索引里的多个 NULL 值将被认为代表多个不同的东西。

除 PRIMARY KEY 以外，其他类型的索引几乎都可以用 CREATE INDEX 语句来添加：

```
CREATE INDEX index_name ON tbl_name (index_columns);
CREATE UNIQUE INDEX index_name ON tbl_name (index_columns);
CREATE FULLTEXT INDEX index_name ON tbl_name (index_columns);
CREATE SPATIAL INDEX index_name ON tbl_name (index_columns);
```

tbl_name、*index_name*和*index_columns*的含义和ALTER TABLE语句里的一样。与ALTER TABLE语句不同的是，CREATE INDEX 语句里的 *index_name* 不是可选的，你也不能用一条 CREATE INDEX 语句创建多个索引。

在使用 CREATE TABLE 语句创建新数据表的同时为它创建索引的语法，和使用 ALTER TABLE 语句添加索引时用的语法很相似，这需要你在定义各数据列的基础上再增加一些索引创建子句：

```
CREATE TABLE tbl_name
(
  ... column definitions ...
  INDEX index_name (index_columns),
  UNIQUE index_name (index_columns),
  PRIMARY KEY (index_columns),
  FULLTEXT index_name (index_columns),
  SPATIAL index_name (index_columns),
  ...
);
```

类似于 ALTER TABLE 语句，*index_name* 在这里也是可选的。如果你省略了它，MySQL 将替你挑选一个。

作为一种特殊情况，你可以通过在某个数据列的定义的末尾加上一条 PRIMARY KEY 或 UNIQUE 子句的办法，来创建一个单数据列的 PRIMARY KEY 或 UNIQUE 索引。比如说，下面两条 CREATE TABLE 语句是等价的：

```
CREATE TABLE mytbl
(
  i INT NOT NULL PRIMARY KEY,
  j CHAR(10) NOT NULL UNIQUE
);

CREATE TABLE mytbl
(
  i INT NOT NULL,
  j CHAR(10) NOT NULL,
  PRIMARY KEY (i),
  UNIQUE (j)
);
```

MEMORY 数据表的默认索引类型是 HASH。利用散列索引进行精确值查询的速度非常快，这也是 MEMORY 数据表的典型用法。不过，如果你打算用一个 MEMORY 数据表进行范围比较（如 id < 100），散列索引的使用效果就没那么理想了。在遇到这类情况时，你最好创建一个 BTREE 索引来代替之，具体做法是在索引定义里增加一条 USING BTREE 子句：

```
CREATE TABLE namelist
(
  id   INT NOT NULL,
  name CHAR(100),
  INDEX USING BTREE (id)
) ENGINE = MEMORY;
```

如果只对某个字符串数据列的一个前缀编索引，在索引定义里命名数据列的语法是 *col_name*(*n*) 而不是简单的 *col_name*。*n* 的含义是索引应该包括数据列值的前 *n* 个字节（二进制字符串类型）或前 *n* 个字符（非二进制字符串类型）。比如说，用下面这条语句创建出来的数据表有一个 CHAR 数据列和一个 BINARY 数据列。它对 CHAR 数据列的前 10 个字符和 BINARY 数据列的前 15 个字节编了索引：

```
CREATE TABLE addresslist
(
  name    CHAR(30) NOT NULL,
  address BINARY(60) NOT NULL,
  INDEX (name(10)),
  INDEX (address(15))
);
```

在对某个字符串数据列的一个前缀编索引时，前缀长度的计量单位必须与该数据列的数据类型相同，就像数据列的长度那样。换句话说，二进制字符串以字节为单位，非二进制字符串以字符为单位。不过，索引项本身的最大长度在 MySQL 内部以字节为单位。这两种长度计量办法对单字节字符集来说没有什么区别，对多字节字符集来说则影响重大。如果某个非二进制字符串包含来自多字节字符集的字符，MySQL 会在索引所允许的字节长度范围内容纳尽可能多的字符。

- 在某些场合，你会发现对数据列前缀（而不是对整个数据列）编索引已经不是你想不想那样做的问题，你只能那样做。这类情况包括以下几种。
- BLOB 或 TEXT 数据列只能创建前缀型索引。
- 索引项本身的长度等于构成索引的各个数据列的索引部分的长度总和。如果这个长度超过了索引项本身所能容纳的最大字节数，你可以通过为数据列前缀编索引来“缩短”这个索引。比如说，假设某个 MyISAM 数据表使用了 latin1 单字节字符集并且包含 4 个 CHAR(255)数据列，数据列的名字是 c1 到 c4。此时，每个数据列的索引项的长度将占用 255 个字节，4 个数据列的索引项的总长度将有 1 020 个字节。可是，因为一个 MyISAM 索引项的最大长度是 1 000 个字节，所以你无法创建一个复合索引把那 4 个数据列的完整内容全部包括进来。解决问题的办法是对部分或全部数据列的一个比较短的前缀部分编索引。比如说，你可以只为前 250 个字符编索引。

FULLTEXT 索引所涉及的数据列是全部带索引的，没有前缀的说法。即使你为 FULLTEXT 索引所涉及的某个数据列指定了一个前缀长度，MySQL 也会忽略它。

可以像下面这样为空间数据类型的数据列（如 POINT 或 GEOMETRY）编索引。

- SPATIAL 索引只能用于 MYISAM 数据表，只能用于 NOT NULL 属性的数据列。所涉及的数据列是全部带索引的。
- 其他索引类型（INDEX、UNIQUE、PRIMNARY KEY）可以配合除 ARCHIVE 以外支持空间数据类型的任何存储引擎使用。不是 PRIMARY KEY 的组成部分的数据列都允许包含 NULL 值。除 POINT 数据列以外，索引所涉及的空间数据列以字节计算的前缀长度必须在创建索引时给出。

3. 删除索引

删除索引的工作可以用 DROP INDEX 或 ALTER TABLE 语句来完成。如果使用的是 DROP INDEX 语句，你必须给出要删除的索引的名字：

```
DROP INDEX index_name ON tbl_name;
```

如果你想用 DROP INDEX 语句删除一个 PRIMARY KEY，就必须以一个带引号的标识符的形式给出

名字 PRIMARY，如下所示：

```
DROP INDEX `PRIMARY` ON tbl_name;
```

这条语句没有任何歧义，因为每个数据表只有一个 PRIMARY KEY，它的名字也永远是 PRIMARY。

类似于 CREATE INDEX 语句，DROP INDEX 在 MySQL 内部将被当做一条 ALTER TABLE 语句来处理。上面这条 DROP INDEX 语句等价于下面两条 ALTER TABLE 语句：

```
ALTER TABLE tbl_name DROP INDEX index_name;
ALTER TABLE tbl_name DROP PRIMARY KEY;
```

如果你不知道某给定数据表的索引的名字，可以用 SHOW CREATE TABLE 或 SHOW INDEX 把它们查出来。

当你从数据表删除数据列时，索引也会隐式地受到影响。如果你删除的数据列是某个索引的组成部分，MySQL 将从索引里删除那个数据列。如果你把构成某个索引的数据列全都丢弃了，MySL 将删除整个索引。

2.6.5　改变数据表的结构

ALTER TABLE 语句的用途非常多。我们在本章前半部分内容里已经见识过它的一些本领了（如改变存储引擎、创建和删除索引等)。你还可以使用 ALTER TABLE 语句来重新命名数据表、添加或删除数据列、改变数据列的数据类型等。本小节只讨论这条语句的一部分功能，附录 E 将描述 ALTER TABLE 语句的完整语法。

当你发现某个数据表的结构不再满足需要时，ALTER TABLE 语句可以帮上你的大忙。比如说，也许是你想用这个数据表来记录一些额外的信息，也许这个数据表所包含的信息已经过时了，也许现有的数据列太小，也许是你当初定义的数据列宽度大到超出了你的需要，而你想把它们缩短一些以节省空间和改善查询性能。下面是 ALTER TABLE 语句会派上大用场的几种情况。

- ❑ 你管理着一个研究项目。你使用了一个 AUTO_INCREMENT 数据列来保存研究记录的编号。你最初的经费不是很充足，最多只能让你生成超过 5 万条记录，所以你决定使用 SMALLINT UNSIGNED 作为该数据列的数据类型，它最多能容纳 65 535 条彼此不重复的记录。没想到，这个项目的研究经费增加了，它现在足以让你再生成 5 万条记录。于是，你需要“加大”那个数据列的类型，以容纳更多的编号。
- ❑ 尺寸调整也可能会朝另一个方向发展。你创建了一个 CHAR(255)数据列，但后来发现数据表里的值没有超过 100 个字节长的。于是，你需要“缩短”那个数据列以节省空间。
- ❑ 你想让某个数据表改用另一种存储引擎以享受该引擎提供的功能。比如说，MyISAM 数据表不具备事务安全性，但你有一个应用程序需要进行事务处理。你可以让有关的数据表改用 InnoDB 或 Falcon 存储引擎，这两种引擎都支持事务处理。

下面是 ALTER TABLE 语句的语法：

```
ALTER TABLE tbl_name action [, action] ... ;
```

每个 action 代表一个你想对数据表进行的修改。有些数据库系统只允许一条 ALTER TABLE 语句完成一个改动，但 MySQL 允许用一条 ALTER TABLE 语句完成多个改动，只要用逗号把它们隔开即可。

提示 如果需要在使用 ALTER TABLE 语句之前查看一下数据表的当前定义，可以执行一条 SHOW CREATE TABLEA 语句。在执行完 ALTER TABLE 语句之后，你还可以用这条语句去验证一下你做的改动对数据表定义的影响是否符合你的预期。

下面的例子演示了 ALTER TABLE 语句的一些典型用法。

改变数据列的数据类型。如果想改变某个数据列的数据类型，可以使用 CHANGE 或 MODIFY 子句。假设 mytbl 数据表里的某个数据列的数据类型是 SMALLINT UNSIGNED，你想把它改成 MEDIUMINT UNSIGNED。下面两条命令都可以达到目的：

```
ALTER TABLE mytbl MODIFY i MEDIUMINT UNSIGNED;
ALTER TABLE mytbl CHANGE i i MEDIUMINT UNSIGNED;
```

为什么在使用 CHANGE 子句时需要写两遍数据列的名字呢？因为 CHANGE 子句能够（而 MODIFY 子句不能）做到的事情是在改变其数据类型的同时重新命名一个数据列。如果想在改变其数据类型的同时把数据列 i 重新命名为 k，你可以这样做：

```
ALTER TABLE mytbl CHANGE i k MEDIUMINT UNSIGNED;
```

在 CHANGE 子句里，需要先给出想改动的数据列的名字，然后给出它的新名字和新定义。因此，即使不想重新命名那个数据列，也需要把它的名字写两遍。

如果只想改变数据列的名字，不改变它的数据类型，先写出 CHANGE old_name new_name、再写出数据列的当前定义即可。

你可以为各数据列指定字符集，具体做法是在数据列的定义里使用 CHARACTER SET 属性来改变它的字符集：

```
ALTER TABLE t MODIFY c CHAR(20) CHARACTER SET ucs2;
```

改变数据类型的重要原因之一，是改善涉及多个数据表中的数据列比较的关联查询的性能。索引通常可以用来在关联查询里比较两个相似的数据列类型，而如果两个数据列的类型完全相同的话，比较的速度会更快。假设你正在运行一个如下所示的查询：

```
SELECT ... FROM t1 INNER JOIN t2 WHERE t1.name = t2.name;
```

如果 t1.name 是 CHAR(10)，t2.name 是 CHAR(15)，查询的速度将不如它们都是 CHAR(15)时那么快。你可以把它们改成相同的，下面两条语句都可以把 t1.name 改成你需要的样子：

```
ALTER TABLE t1 MODIFY name CHAR(15);
ALTER TABLE t1 CHANGE name name CHAR(15);
```

让数据表改用另一种存储引擎。如果想让数据表改用另一种存储引擎，用 ENGINE 子句给出新存储引擎的名字就行了：

```
ALTER TABLE tbl_name ENGINE = engine_name;
```

engine_name 是一个诸如 MyISAM、MEMORY 或 InnoDB 之类的名字，不区分字母的大小写。

让数据表改用另一种存储引擎的常见原因是让它具备事务安全性。假设你有一个 MyISAM 数据表，而一个用到了这个数据表的应用程序需要进行事务操作，包括意外故障发生时的事务回滚机制。MyISAM 数据表不支持事务处理，但你可以通过把它转换为一个 InnoDB 或 Falcon 数据表让它具备事务安全性：

```
ALTER TABLE tbl_name ENGINE = InnoDB;
ALTER TABLE tbl_name ENGINE = Falcon;
```

当你打算让数据表改用另一种存储引擎时，能否达到你的目的还要取决于新老两种存储引擎的功能是否兼容。例如，以下转换就是不允许的。

- 如果你有一个数据表包含着一个 BLOB 数据列，你将不能把它转换为使用 MEMORY 引擎，因为 MEMORY 引擎不支持 BLOB 数据列。
- 如果你有一个 MyISAM 数据表包含着 FULLTEXT 或 SPATIAL 索引，你将不能把它转换为使用另一种引擎，因为只有 MyISAM 支持这两种索引。

在以下条件下，你不应该使用 ALTER TABLE 语句让数据表改用另一种存储引擎。

- MEMORY 数据表存在于内存中，在服务器退出运行时将消失。因此，如果你希望某个数据表的内容在服务器重新启动后仍然存在，就不应该把它转换为 MEMORY 类型。
- 如果你使用了一个 MERGE 数据表来管理一组 MyISAM 数据表，就应该避免使用 ALTER TABLE 语句去改变个别 MyISAM 数据表的结构，除非你决定对所有的成员 MyISAM 数据表和那个 MERGE 数据表做出同样的修改。MERGE 数据表的正常使用需要其全体成员 MyISAM 数据表有着同样的结构。
- InnoDB 数据表可以被转换为使用另一种存储引擎。不过，如果你为你的 InnoDB 数据表定义了外键约束条件，那些约束条件在转换后将不复存在，因为只有 InnoDB 才支持外键。

重新命名一个数据表。用 RENAME 子句给数据表起一个新名字：

```
ALTER TABLE tbl_name RENAME TO new_tbl_name;
```

另一个办法是使用 RENAME TABLE 语句来重新命名数据表。下面是它的语法：

```
RENAME TABLE old_name TO new_name;
```

ALTER TABLE 语句每次只能重新命名一个数据表，而 RENAME TABLE 语句可以一次重新命名多个数据表。比如说，你可以像下面这样交换两个数据表的名字：

```
RENAME TABLE t1 TO tmp, t2 TO t1, tmp TO t2;
```

如果在重新命名一个数据表时在它的名字前面加上了数据库名前缀，就可以把它从一个数据库移动到另一个数据库。下面两条语句都可以把数据表 t 从 sampdb 数据库移到 test 数据库去：

```
ALTER TABLE sampdb.t RENAME TO test.t;
RENAME TABLE sampdb.t TO test.t;
```

不能把一个数据表重新命名为一个已有的名字。

如果重新命名的某个 MyISAM 数据表是某个 MERGE 数据表的成员，你必须重新定义那个 MERGE 数据表，让它使用那个 MyISAM 数据表的新名字。

2.7　获取数据库的元数据

MySQL 提供了好几种办法供我们获取关于数据库和数据库里各种对象（也就是数据库的元数据）的信息：

- 各种 SHOW 语句，如 SHOW DATABASES 或 SHOW TABLES；
- INFORMATION_SCHEMA 数据库里的数据表；
- 命令行程序，如 mysqlshow 或 mysqldump。

接下来的几节描述使用这些信息来源去访问元数据的具体做法。

2.7.1 用SHOW语句获取元数据

MySQL 提供的 SHOW 语句能够以多种方式显示数据库的元数据。SHOW 语句可以帮助我们及时了解数据库内容的变化情况，查看各有关数据表的结构。下面的例子演示了 SHOW 语句的几种典型用法。

列出服务器所管理的数据库：

```
SHOW DATABASES;
```

查看给定数据库的 CREATE DADABASE 语句：

```
SHOW CREATE DATABASE db_name;
```

列出默认数据库或给定数据库里的数据表：

```
SHOW TABLES;
SHOW TABLES FROM db_name;
```

SHOW TABLES 语句的输出报告里不包括 TEMPORARY 数据表。

查看数据表的 CREATE TABLE 语句：

```
SHOW CREATE TABLE tbl_name;
```

查看数据表里的数据列或索引信息：

```
SHOW COLUMNS FROM tbl_name;
SHOW INDEX FROM tbl_name;
```

DESCRIBE tbl_name 和 EXPLAIN tbl_name 语句是 SHOW COLUMNS FROM tbl_name 语句的同义语句。

查看默认数据库或某给定数据库里的数据表的描述性信息：

```
SHOW TABLE STATUS;
SHOW TABLE STATUS FROM db_name;
```

有几种 SHOW 语句还可以带有一条 LIKE 'pattern'子句，这个子句用来把 SHOW 语句的输出限制在给定范围。MySQL 将把'pattern'表达式解释为 SQL 模式，这种模式允许包含“%”和“_”通配符。比如说，下面这条语句可以把 student 数据表里名字以字符“s”开头的数据列查出来：

```
mysql> SHOW COLUMNS FROM student LIKE 's%';
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| sex        | enum('F','M')    | NO   |     |         |                |
| student_id | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
+------------+------------------+------+-----+---------+----------------+
```

如果需要在 LIKE 模式里实际使用某个通配符，必须在该字符的前面加上一个反斜线进行转义。这种情况常见于需要匹配“_”（下划线）字符时，该字符经常出现在数据库、数据表和数据列的名字里。

支持 LIKE 子句的所有 SHOW 语句都可以被改写为使用一条 WHERE 子句。WHERE 子句不影响 SHOW 语句的输报告里的数据列个数，它只改变 SHOW 语句输出的数据行的个数。WHERE 子句应该引用 SHOW 语句将输出的数据列。如果某个数据列的名字是一个保留字（如 KEY），就必须以带引号的标识符的方

式给出。下面这条语句用来确定 student 数据表里的主键是哪一个数据列：

```
mysql> SHOW COLUMNS FROM student WHERE `Key` = 'PRI';
+------------+------------------+------+-----+---------+----------------+
| Field      | Type             | Null | Key | Default | Extra          |
+------------+------------------+------+-----+---------+----------------+
| student_id | int(10) unsigned | NO   | PRI | NULL    | auto_increment |
+------------+------------------+------+-----+---------+----------------+
```

有时候，如何能从一个应用程序里检查某个给定的数据表是否存在，这将很有用。可以用 SHOW TABLES 语句来达到这个目的（但别忘了 SHOW TABLES 语句的输出里不包括 TEMPORARY 数据表）：

```
SHOW TABLES LIKE 'tbl_name';
SHOW TABLES FROM db_name LIKE 'tbl_name';
```

如果 SHOW TABLES 列出了那个数据表的信息，就说明它存在。你还可以通过下面两条语句中的一条去检查某给定数据表是否存在，它们可以把 TEMPORARY 数据表也找出来：

```
SELECT COUNT(*) FROM tbl_name;
SELECT * FROM tbl_name WHERE FALSE;
```

如果数据表存在，这两条语句都将执行成功；如果不存在，两条语句都将失败。第一条语句最适用于 MyISAM 数据表，因为不带 WHERE 子句的 COUNT(*)函数已经得到了高度的优化。它不太适用于 InnoDB 数据表，因为对 InnoDB 数据表里的数据行的总数进行统计将导致一次全表扫描。第二条语句更通用一些，它在任何一种存储引擎下都执行得很快。这些语句最适合用在 Perl 或 PHP 等应用程序设计语言里，因为你可以测试你的查询是成功还是失败并采取相应的行动。它们不太适合用在你打算通过 mysql 工具运行的批处理脚本里，这是因为你在发生错误时除了让脚本退出运行以外不能做任何事情。（当然，你也可以忽略那个错误。但在发生错误之后，还有必要继续进行你的查询吗？）

如果你想知道各个数据表使用的是哪一种存储引擎，可以使用 SHOW TABLE STATUS 或 SHOW CREATE TABLE 语句。在这两条语句的输出报告里都有关于存储引擎指示的信息。

2.7.2　从INFORMATION_SCHEMA数据库获取元数据

获取数据库元信息的另一个办法是访问 INFORMATION_SCHEMA 数据库。INFORMATION_SCHEMA 数据库以 SQL 语言标准为基础。换句话说，对 INFORMATION_SCHEMA 数据库的访问机制是标准化的，虽然有一部分内容是 MySQL 特有的。这使得 INFORMATION_SCHEMA 数据库有着优于各种 SHOW 语句的可移植性，那些语句都是 MySQL 专用的。

对 INFORMATION_SCHEMA 数据库的访问需要通过 SELECT 语句来进行，所以这种访问可以非常灵活。在 SHOW 语句的输出报告里，数据列的个数是固定的，而且你无法把那些输出捕获到一个数据表里去。INFORMATION_SCHEMA 数据库就不同了，SELECT 语句可以选取特定的数据列进入输出报告，WHERE 子句可以让你通过各种表达式挑选你真正需要的信息。不仅如此，你还可以使用联结或子查询，可以使用 CREATE TABLE...SELECT 或 INSERT INTO...SELECT 语句把检索结果保存到另一个数据表做进一步处理。

可以把 INFORMATION_SCHEMA 数据库想象成一个虚拟的数据库，这个数据库里的数据表是一些由不同的数据库元数据构成的视图。如果你想知道 INFORMATION_SCHEMA 数据库都包含哪些数据表，请使用 SHOW TABLES 命令。下面的输出报告取材于 MySQL 5.1（5.0 版的数据表要少一些）。

```
mysql> SHOW TABLES IN INFORMATION_SCHEMA;
+---------------------------------------+
| Tables_in_information_schema          |
+---------------------------------------+
| CHARACTER_SETS                        |
| COLLATIONS                            |
| COLLATION_CHARACTER_SET_APPLICABILITY |
| COLUMNS                               |
| COLUMN_PRIVILEGES                     |
| ENGINES                               |
| EVENTS                                |
| FILES                                 |
| GLOBAL_STATUS                         |
| GLOBAL_VARIABLES                      |
| KEY_COLUMN_USAGE                      |
| PARTITIONS                            |
| PLUGINS                               |
| PROCESSLIST                           |
| REFERENTIAL_CONSTRAINTS               |
| ROUTINES                              |
| SCHEMATA                              |
| SCHEMA_PRIVILEGES                     |
| SESSION_STATUS                        |
| SESSION_VARIABLES                     |
| STATISTICS                            |
| TABLES                                |
| TABLE_CONSTRAINTS                     |
| TABLE_PRIVILEGES                      |
| TRIGGERS                              |
| USER_PRIVILEGES                       |
| VIEWS                                 |
+---------------------------------------+
```

下面是对各个 INFORMATION_SCHEMA 数据表的简要说明。

- SCHEMATA、TABLES、VIWS、ROUTINES、TRIGGERS、EVENTS、PARTITIONS、COLUMNS。关于数据库、数据表、视图、存储例程（stored routine）、触发器、数据库里的事件、数据表分区和数据列的信息。
- FILES。关于 NDB 硬盘数据文件的信息。
- TABLE_CONSTRAINS、KEY_COLUMN_USAGE。关于数据表和数据列上的约束条件的信息，如唯一化索引、外键等。
- STATISTICS。关于数据表索引特性的信息。
- REFERENTIAL_CONSTRAINS。关于外键的信息。
- CHARACTER_SETS、COLLATIONS、COLLATION_CHARACTER_SET_APPLICABILITY。关于所支持的字符集、每种字符集的排序方式、每种排序方式与它的字符集的映射关系的信息。
- ENGINES、PLUGINS。关于存储引擎和服务器插件的信息。
- USER_PRIVILEGES、SCHEMA_PRIVILEGES、TABLE_PRIVILEGES、COLUMN_PRIVILEGES。全局、数据库、数据表和数据列的权限信息，这些信息分别来自 mysql 数据库里的 user、db、tables_priv、column_priv 数据表。

- ❑ GLOBAL_VARIABLES、SESSION_VARIABLES、GLOBAL_STATUS、SESSION_STATUS。全局和会话级系统变量和状态变量的值。
- ❑ PROCESSLIST。关于在服务器内执行的线程的信息。

各个存储引擎还会在 INFORMATION_SCHEMA 数据库里增加它们专用的数据表。Falcon 存储引擎就是如此（如果它们已被激活）。

如果你想知道某给定 INFORMATION_SCHEMA 数据表都包含有哪些数据列，可以使用 SHOW COLLUMNS 或 DESCRIBE 语句去查看：

```
mysql> DESCRIBE INFORMATION_SCHEMA.CHARACTER_SETS;
+----------------------+-------------+------+-----+---------+-------+
| Field                | Type        | Null | Key | Default | Extra |
+----------------------+-------------+------+-----+---------+-------+
| CHARACTER_SET_NAME   | varchar(64) | NO   |     |         |       |
| DEFAULT_COLLATE_NAME | varchar(64) | NO   |     |         |       |
| DESCRIPTION          | varchar(60) | NO   |     |         |       |
| MAXLEN               | bigint(3)   | NO   |     | 0       |       |
+----------------------+-------------+------+-----+---------+-------+
```

如果你想查看某个 INFORMATION_SCHEMA 数据表里的信息,请使用 SELECT 语句。（INFORMATION_SCHEMA 数据库以及它里面的所有数据表和数据列的名字都不区分字母的大小写。）查看某给定 INFORMATION_SCHEMA 数据表里的所有数据列的通用查询语法如下所示：

```
SELECT * FROM INFORMATION_SCHEMA.tbl_name;
```

如果你想有选择地查看数据库元信息，加上一条相应的 WHERE 子句即可。

前面介绍了如何利用 SHOW 语句去检查某个数据表是否存在以及它使用的是哪一种存储引擎。INFORMATION_SCHEMA 数据表可以提供同样的信息。下面的查询利用 INFORMATION_SCHEMA 数据表去测试某个特定的数据表是否存在，如果那个数据表存在，返回 1；如果它不存在，返回 0。

```
mysql> SELECT COUNT(*) FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA='sampdb' AND TABLE_NAME='member';
+----------+
| COUNT(*) |
+----------+
|        1 |
+----------+
```

下面这个查询可以让你知道某个数据表使用的是哪一种存储引擎：

```
mysql> SELECT ENGINE FROM INFORMATION_SCHEMA.TABLES
    -> WHERE TABLE_SCHEMA='sampdb' AND TABLE_NAME='student';
+--------+
| ENGINE |
+--------+
| InnoDB |
+--------+
```

2.7.3　从命令行获取元数据

mysqlshow 命令提供的信息与相应的 SHOW 语句相同或近似，这使我们可以从我们的命令行提示符获取数据库和数据表信息。

服务器所管理的数据库：

```
% mysqlshow
```

列出某给定数据库里的数据表：

```
% mysqlshow db_name
```

查看某给定数据表里的数据列信息：

```
% mysqlshow db_name tbl_name
```

查看某给定数据表里的索引信息：

```
% mysqlshow --keys db_name tbl_name
```

查看某给定数据库里的数据表的描述性信息：

```
% mysqlshow --status db_name
```

mysqldump 客户程序能够让你看到 CREATE TABLE 语句（就像 SHOW CREATE TABLE 语句那样）所定义的数据表结构。在使用 mysqldump 程序去查看数据表结构时，千万不要忘记加上--no-data 选项，否则你看到的将是数据表里的数据！

```
% mysqldump --no-data db_name [tbl_name] ...
```

如果你只给出了数据库的名字而没有给出任何数据表的名字，mysqldump 将把该数据库里的所有数据表的结构呈现在你眼前。否则，它将只显示有名字的数据表的结构信息。

在使用 mysqlshow 和 mysqldump 程序时，不要忘记给出必要的连接参数选项，如--host、--user 或--password。

2.8 利用联结操作对多个数据表进行检索

如果只把数据存入数据库，而不对它们进行检索或是用它们来干些什么，这是没有什么意义的。那正是 SELECT 语句的目的所在：帮你找到你需要的数据。与 SQL 语言中的其他语句相比，SELECT 语句应该是最常用的，同时也应该是最不容易掌握的。用来筛选数据行的条件有时会非常复杂并需要比较多个数据表里的数据列。

SELECT 语句的基本语法如下所示：

```
SELECT select_list          # What columns to select
FROM table_list             # The tables from which to select rows
WHERE row_constraint        # What conditions rows must satisfy
GROUP BY grouping_columns # How to group results
ORDER BY sorting_columns    # How to sort results
HAVING group_constraint     # What conditions groups must satisfy
LIMIT count;                # Row count limit on results
```

除了用来表明你想得到什么样的输出的 SELECT 和 select_list 部分，这个语法中的所有东西都是可选的。有些数据库软件要求 FROM 子句也必不可少，但 MySQL 没有那样做，这使你可以在没有引用任何数据表的情况下对表达式进行求值：

```
SELECT SQRT(POW(3,2)+POW(4,2));
```

在第 1 章里，我们重点讨论了只涉及一个数据表的 SELECT 语句，把注意力主要集中在了输出数据列清单和 WHERE、GROUP BY、HAVING 和 LIMIT 子句上。本节将讨论 SELECT 语句最不容易掌握的

方面：编写联结查询，也就是编写从多个数据表检索数据的 SELECT 语句。我们将讨论 MySQL 所支持的联结操作的类型、它们的含义、如何使用它们等。这将帮助你更有效率地用好 MySQL。在许多时候，写出正确的查询命令的关键其实就是正确地把数据表联结在一起。

在使用SELECT语句时经常遇到这样一种情况：那是你一个从没见过的新问题，而编写一个SELECT查询去解决它并不总是那么轻而易举。不过，一旦你把它解决了，以后再遇到类似的问题时就有经验了。只有具备足够经验才能最有效地利用 SELECT 语句，原因很简单，需要用它来解决的问题实在是千变万化。

随着经验的积累，你会发现有很多新问题都可以通过联结操作轻松解决，你会发现自己在这样考虑："噢，这是一个左联结问题。"或者："啊哈，用一个三方联结，它加上一个键字数据列配对限制就可以搞定。"（我希望"经验可以帮上大忙"这句话让你受到鼓舞，否则你发现自己很难用那些古怪的术语思维。）

在接下来的讨论里，有许多用来演示如何使用 MySQL 所支持的联结操作的例子需要用到如下所示的 t1 和 t2 数据表：

```
数据表 t1        数据表 t2
+----+----+     +----+----+
| i1 | c1 |     | i2 | c2 |
+----+----+     +----+----+
|  1 | a  |     |  2 | c  |
|  2 | b  |     |  3 | b  |
|  3 | c  |     |  4 | a  |
+----+----+     +----+----+
```

之所以选择这样两个数据表是因为它们都很小，可以让我们看到联结操作的完整结果。

涉及多个数据表的 SELECT 语句的其他类型是子查询（嵌套在另一个 SELECT 语句里的 SELECT 语句）和 UNION 语句。我们将在 2.9 节和 2.10 节中分别讨论它们。

MySQL 支持的、与涉及多个数据表的检索操作密切相关的其他功能，包括根据某个数据表的内容来删除或刷新另一个数据表里的数据行。比如说，你可能需要从某个数据表里把它在另一个数据表里没有任何匹配的记录全部删掉，或者需要从某个数据表里把几个数据列复制到另一个数据表里去。2.12 节将讨论这些问题。

2.8.1　内联结

如果你在 SELECT 语句的 FROM 子句里列出了多个数据表，并用 INNER JOIN 将它们的名字隔开，MySQL 将执行*内联结*（inner join）操作，这将通过把一个数据表里的数据行与另一个数据表里的数据行进行匹配来产生结果。比如说，如果你像下面这样把 t1 和 t2 联结起来，t1 里的每一个数据行将与 t2 里的每一个数据行组合：

```
mysql> SELECT * FROM t1 INNER JOIN t2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  1 | a  |  2 | c  |
|  2 | b  |  2 | c  |
|  3 | c  |  2 | c  |
|  1 | a  |  3 | b  |
```

```
|  2 | b  |  3 | b  |
|  3 | c  |  3 | b  |
|  1 | a  |  4 | a  |
|  2 | b  |  4 | a  |
|  3 | c  |  4 | a  |
+----+----+----+----+
```

在这条语句里，SELECT *的含义是，从 FROM 子句所列出的每一个数据表里选取每一个数据列。你也可以把这写成 SELECT t1.*, t2.*：

```
SELECT t1.*, t2.* FROM t1 INNER JOIN t2;
```

如果你要选取所有的数据列，或者想按另外一种从左到右的顺序来显示它们，按照你希望的顺序依次列出各数据列（记得用逗号把它们隔开）就行了。

根据某个数据表里的每一个数据行与另一个数据表里的每一个数据行得到全部可能的组合的联结操作叫做生成*笛卡儿积*（cartesian product）。像这样来联结数据表有可能产生非常多的结果数据行，因为结果数据行的总数将是对每个数据表的数据行个数进行连续乘法而得到的积。假设你有 3 个数据表分别包含 100、200 和 300 个数据行，它们的交叉联结将返回 6 百万个（100 × 200 × 300）数据行。这可是个相当大的数字，而那 3 个数据表都很小。在这种情况下，通常需要增加一条 WHERE 子句以便把结果集压缩到一个更便于管理的尺寸。

如果你增加了一条如下所示的 WHERE 子句，让联结操作只对各数据表里的特定数据列里的值进行匹配，联结操作将只选取那些数据列里的值彼此相等的数据行：

```
mysql> SELECT t1.*, t2.* FROM t1 INNER JOIN t2 WHERE t1.i1 = t2.i2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  2 | b  |  2 | c  |
|  3 | c  |  3 | b  |
+----+----+----+----+
```

CROSS JOIN 和 JOIN 联结类型类似于 INNER JOIN。比如说，下面的语句是等价的：

```
SELECT t1.*, t2.* FROM t1 INNER JOIN t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 CROSS JOIN t2 WHERE t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t1 JOIN t2 WHERE t1.i1 = t2.i2;
```

“,”（逗号）关联操作符的效果与 INNER JOIN 相似：

```
SELECT t1.*, t2.* FROM t1, t2 WHERE t1.i1 = t2.i2;
```

请注意，逗号操作符的优先级和其他联结类型不一样，有时还会导致语法错误，而其他联结类型没有这些问题。我个人认为应该尽量避免使用逗号操作符。

INNER JOIN、CROSS JOIN 和 JOIN（注意，不包括逗号操作符）还支持另外几种用来表明如何对数据表里的数据列进行匹配的语法变体，如下所示。

- 用一条 ON 子句代替 WHERE 子句。下面是一个使用了 ON 子句的 INNER JOIN 操作的例子：

  ```
  SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ON t1.i1 = t2.i2;
  ```

  不管被联结的数据列是否同名，ON 子句都可以使用。

- 另一种语法是使用一个 USING()子句，它在概念上类似于 ON 子句，但要求被联结的数据列必须是同名的。比如说，如下所示的查询对 mytbl1.b 和 mytbl2.b 进行了联结：

```
SELECT mytbl1.*, mytbl2.* FROM mytbl1 INNER JOIN mytbl2 USING (b);
```

2.8.2　避免歧义：如何在联结操作中给出数据列的名字

在 SELECT 语句里给出的数据列的名字不允许产生歧义，而且必须来自 FROM 子句里的某个数据表。如果只涉及一个数据表，就不会产生歧义，被列出的所有数据列都来自那个数据表。如果涉及多个数据表，只在一个表中出现的数据列名称不会产生歧义。不过，如果某个数据列的名字出现在多个数据表里，在引用这个数据列时必须使用 tbl_name1.col_name 语法给它加上一个数据表的名字来表明它来自哪一个数据表。假设你的 mytbq1 数据表里包含数据列 a 和 b，mytbl2 数据表里包含数据列 b 和 c。对于这种情况，引用 a 或 c 都不会产生歧义，但在引用 b 时就必须把它限定为 mytbl1.b 或 mytbl2.b 了：

```
SELECT a, mytbl1.b, mytbl2.b, c FROM mytbl1 INNER JOIN mytbl2 ... ;
```

有时候，用数据表的名字进行限定仍不足以解决数据列的歧义问题。比如说，假设你正在进行一个自联结操作（也就是把一个数据表与它本身联结起来），这是在查询命令里多次用到同一个数据表（而不是用到多个数据表）的问题，用数据表的名字来限定数据列帮不上你的忙。在遇到这类问题时，数据表别名可以让你达到目的。你只需给该数据表的某个实例起一个别名，就可以通过 alias_name1.col_name 语法来引用该实例里的数据列了。下面的查询命令将把一个数据表与它自身联结起来。为了消除在引用数据列时可能产生的歧义问题，我们给该数据表的一个实例分配了一个别名：

```
SELECT mytbl.col1, m.col2 FROM mytbl INNER JOIN mytbl AS m
WHERE mytbl.col1 > m.col1;
```

2.8.3　左联结和右联结（外联结）

内联结只显示在两个数据表里都能找到匹配的数据行。外联结除了显示同样的匹配结果，还可以把其中一个数据表在另一个数据表里没有匹配的数据行也显示出来。外联结分左联结和右联结两种。本小节里的绝大多数例子使用的是 LEFT JOIN，意思是把左数据表在右数据表里没有匹配的数据行也显示出来。RIGHT JOIN 刚好与此相反，它将把右数据表在左数据表里没有匹配的数据行也显示出来，数据表的角色调换了一下。

LEFT JOIN 的工作情况是这样的：你给出用来匹配两个数据表里的数据行的数据列，当来自左数据表的某个数据行与来自右数据表的某个数据行匹配时，那两个数据行的内容就会被选取为一个输出数据行；如果来自左数据表的某个数据行在右数据表里找不到匹配，它也会被选取为一个输出数据行，此时与它联结的是一个来自右数据表的“假”数据行，这个“假”数据行的所有数据列都包含 NULL 值。

换句话说，在 LEFT JOIN 操作里，来自左数据表的每一个数据行在结果集里都有一个对应的数据行，不管它在右数据表里有没有匹配。在结果集里，在右数据表里没有匹配的结果数据行有这样一个特征：来自右数据表的所有数据列都是 NULL 值。这个特征可以让你知道右数据表里缺少了哪些数据行。这是一个既有趣又重要的特征，能够反映出各种各样的问题。还没有为哪些顾客指派服务代表？哪些库存商品一件也没卖出去？或者，就我们的 sampdb 数据库而言，哪些学生没有参加过某次特定的考试？哪些学生在 absence 数据表里没有任何记录（也就是说，哪些学生是全勤的）？

我们仍以刚才的 t1 和 t2 数据表为例：

```
数据表 t1        数据表 t2
+----+----+      +----+----+
| i1 | c1 |      | i2 | c2 |
+----+----+      +----+----+
|  1 | a  |      |  2 | c  |
|  2 | b  |      |  3 | b  |
|  3 | c  |      |  4 | a  |
+----+----+      +----+----+
```

如果我们对这两个数据表进行内联结以匹配 t1.i1 和 t2.i2，我们只能得到与值 2 和 3 相匹配的输出，因为只有它们才是这两个数据表里都有的值：

```
mysql> SELECT t1.*, t2.* FROM t1 INNER JOIN t2 ON t1.i1 = t2.i2;
+----+----+----+----+
| i1 | c1 | i2 | c2 |
+----+----+----+----+
|  2 | b  |  2 | c  |
|  3 | c  |  3 | b  |
+----+----+----+----+
```

左联结操作会为 t1 数据表里每一个数据行生成一个输出数据行，不管它在 t2 里有没有匹配。为了写出相应的左联结查询命令，只要在上面这条语句里把两个数据表之间的 INNER JOIN 替换为 LEFT JOIN 即可：

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
+----+----+------+------+
| i1 | c1 | i2   | c2   |
+----+----+------+------+
|  1 | a  | NULL | NULL |
|  2 | b  |    2 | c    |
|  3 | c  |    3 | b    |
+----+----+------+------+
```

请注意，结果集里多出了一个 t1.i1 取值为 1 的输出行，它在 t2 里没有任何匹配。在这个输出行里，来自 t2 的所有数据列的值都是 NULL。

在使用 LEFT JOIN 时需要注意这样一个问题：如果右数据表里的数据列没有被全部定义成 NOT NULL，结果集里的数据行就可能不能反映真实情况。比如说，如果右数据表里的某个数据列允许取值为 NULL 并被收录在结果集里，用这个数据列里的 NULL 值来判断“在右数据表里没有匹配”就可能出问题。

正如刚才提到的那样，RIGHT JOIN 和 LEFT JOIN 的行为相似，只是把数据表的角色调换了一下而已。下面两条语句是等价的：

```
SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2;
SELECT t1.*, t2.* FROM t2 RIGHT JOIN t1 ON t1.i1 = t2.i2;
```

下面的讨论将围绕 LEFT JOIN 展开，但同样适用于 RIGHT JOIN，只要调换一下数据表的角色就行了。

LEFT JOIN 很有用，尤其是在你只想找出在右数据表里没有匹配的左数据表的行时。具体地说，增加一条 WHERE 子句，让它把右数据表的数据列全部是 NULL 值的（也就是那些在一个数据表里有匹配，但在另一个数据表里没有匹配）数据行筛选出来：

```
mysql> SELECT t1.*, t2.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
    -> WHERE t2.i2 IS NULL;
+----+----+------+------+
| i1 | c1 | i2   | c2   |
+----+----+------+------+
|  1 | a  | NULL | NULL |
+----+----+------+------+
```

一般来说，在编写这样的查询命令时，你真正感兴趣的东西是左数据表里没被匹配的值。把来自右数据表值为 NULL 的数据列显示出来没有什么意义，你可以把它们从将被输出的数据列清单里剔除：

```
mysql> SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.i1 = t2.i2
    -> WHERE t2.i2 IS NULL;
+----+----+
| i1 | c1 |
+----+----+
|  1 | a  |
+----+----+
```

类似于 INNER JOIN，LEFT JOIN 也可以写成使用一个 ON 子句或一个 USING()子句来给出匹配条件。与 INNER JOIN 的情况相同，ON 子句不管被联结的数据列是否同名都可以使用，USING()子句要求被联结的数据列必须有同样的名字。

LEFT JOIN 还有几种同义词和变体。LEFT OUTER JOIN 是 LEFT JOIN 的一个同义词。MySQL 还支持一种 ODBC 风格的 LEFT OUTER JOIN 表示法，该表示法使用了花括号和 OJ（outer join）：

```
mysql> SELECT t1.* FROM { OJ t1 LEFT OUTER JOIN t2 ON t1.i1 = t2.i2 }
    -> WHERE t2.i2 IS NULL;
+----+----+
| i1 | c1 |
+----+----+
|  1 | a  |
+----+----+
```

NATURAL LEFT JOIN 类似于 LEFT JOIN。它将按照 LEFT JOIN 规则对左、右数据表里所有同名的数据列进行匹配。（也就是说，它不需要你给出任何 ON 或 USING 子句。）

正如刚才提到的那样，LEFT JOIN 非常适合用来解决“哪些值是缺失的”这个问题。接下来，我们将把这个原则应用到 sampdb 数据库上，来考虑一个比刚才那些使用 t1 和 t2 数据表的例子更复杂的例子。

对于记录学生学习情况的项目（请参见第 1 章），我们有一个 student 数据表来记录学生、一个 grade_event 数据表来记录已经发生的考试或测验事件，还有一个 score 数据表来列出每位学生的每次考试或测验成绩。如果某个学生在某次考试或小测验的当天生病了，score 数据表里就不会有该名学生在那次事件的成绩。因故误考的学生需要参加补考，可怎样才能把这些缺失的数据行找出来呢？

解决问题的关键是找出哪些学生在某次考试里没有任何成绩，这种查找需要为每一个考试事件执行一遍。换个技术味儿更浓的说法，我们需要找出哪些学生和事件的组合在 score 数据表里没有出现过。这种“哪些值没有出现过”的说法是需要对数据库进行 LEFT JOIN 查询的一种标志。这个联结查询可不像前几个例子那么简单：我们这次要找的不是哪个值没有出现在某个数据列，我们要找的是一些由两个数据列构成的组合。我们想要的组合是所有的“学生+事件”，这可以通过把 student 数据表

和 grade_event 数据表联结起来得到：

```
FROM student INNER JOIN grade_event
```

接下来，我们需要把这个联结结果与 score 数据表用一个 LEFT JOIN 操作联结起来以找出匹配的学生 ID/考试 ID 组合：

```
FROM student INNER JOIN grade_event
     LEFT JOIN score ON student.student_id = score.student.id
                     AND grade_event.event_id = score.event_id
```

请注意 ON 子句，score 数据表里的数据行将按照 ON 子句给出的匹配条件与刚才那个 INNER JOIN 操作的结果集进行左联结。这是解决这个问题的关键。这次 LEFT JOIN 操作将为对 student 和 grade_event 数据表进行内联结而得到的每一个数据行生成一个数据行，即使没有对应的 score 表里的行。在这次 LEFT JOIN 的结果集里，缺少考试分数的数据行可以根据来自 score 数据表的数据列将全部是 NULL 值这一事实被找出来。我们可以通过在 WHERE 子句里增加一个条件的办法把这些数据行找出来。来自 score 数据表的任何数据列都可以用在这个 WHERE 条件里，但因为我们正在寻找的是缺少考试分数的数据行，对 score 数据列进行测试当然最顺理成章：

```
WHERE score.score IS NULL
```

我们还可以用一个 ORDER BY 子句对结果进行排序。有两种顺序最符合逻辑：按每个学生参加的考试和按参加每次考试的学生，我选的是前者：

```
ORDER BY student.student_id, grade_event.event_id
```

现在，只需列出我们想在输出报告里看到的数据列就可以大功告成了。下面是最终的 SELECT 语句：

```
SELECT
  student.name, student.student_id,
  grade_event.date, grade_event.event_id, grade_event.category
FROM
  student INNER JOIN grade_event
  LEFT JOIN score ON student.student_id = score.student_id
                  AND grade_event.event_id = score.event_id
WHERE
  score.score IS NULL
ORDER BY
  student.student_id, grade_event.event_id;
```

下面是运行这个查询所得到的结果：

```
+-----------+------------+------------+----------+----------+
| name      | student_id | date       | event_id | category |
+-----------+------------+------------+----------+----------+
| Megan     |          1 | 2008-09-16 |        4 | Q        |
| Joseph    |          2 | 2008-09-03 |        1 | Q        |
| Katie     |          4 | 2008-09-23 |        5 | Q        |
| Devri     |         13 | 2008-09-03 |        1 | Q        |
| Devri     |         13 | 2008-10-01 |        6 | T        |
| Will      |         17 | 2008-09-16 |        4 | Q        |
| Avery     |         20 | 2008-09-06 |        2 | Q        |
| Gregory   |         23 | 2008-10-01 |        6 | T        |
| Sarah     |         24 | 2008-09-23 |        5 | Q        |
```

```
| Carter     |         27 | 2008-09-16 |         4 | Q        |
| Carter     |         27 | 2008-09-23 |         5 | Q        |
| Gabrielle  |         29 | 2008-09-16 |         4 | Q        |
| Grace      |         30 | 2008-09-23 |         5 | Q        |
+-----------+------------+------------+----------+----------+
```

这里还有一个小细节需要解释一下。我们可以在输出报告里看到 student_id 和 event_id。student_id 数据列在 student 和 score 数据表里都有，所以你们也许会认为把输出列命名为 student.student_id 或 score.student_id 没什么区别。事实却并非如此：我们之所以能找到我们感兴趣的数据行，是因为那个 LEFT JOIN 操作所返回的 score 数据表里的数据列全部都是 NULL 值。如果选择 score.student_id，在输出报告里就会有一个全部都是 NULL 值的数据列。在决定显示来自哪个数据表的 event_id 数据列时也需要考虑这个因素：它在 grade_event 和 score 数据表里都出现了，但因为 score.event_id 值将全部是 NULL，所以在查询命令里需要选择 grade_event.event_id 作为输出列。

2.9　用子查询进行多数据表检索

MySQL 支持子查询（subquery），也就是把一条 SELECT 语句用括号括起来嵌入另一个 SELECT 语句。下面这个例子从 grade_event 数据表里找出对应于考试（'T'）的 event_id，再用它们去选取那些考试的成绩：

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM grade_event WHERE category = 'T');
```

子查询可以返回以下不同类型的信息。

- 标量子查询将返回一个值。
- 数据列子查询将返回一个由一个或多个值构成的数据列。
- 数据行子查询将返回一个由一个或多个值构成的数据行。
- 数据表子查询将返回一个由一个或多个数据行构成的数据表，数据行可以由一个或多个数据列构成。

子查询的结果可以用以下不同的办法进行测试。

- 标量子查询的结果可以用诸如“=”或“<”之类的相对比较操作符进行求值。
- 可以用 IN 和 NOT IN 操作符来测试某给定值是否包含在子查询的结果集里。
- 可以用 ALL、ANY 和 SOME 操作符把某给定值与子查询的结果集进行比较。
- 可以用 EXISTS 和 NOT EXISTS 操作符来测试子查询的结果集是否为空。

标量子查询是最严格的，它只产生一个值。也正是因为如此，标量子查询的适用范围也最大。从理论上讲，标量子查询可以出现在允许使用标量操作数的任何地方，你可以把它用做表达式里的一个操作数或函数的输入参数，还可以让它们出现在输出数据列的清单里。数据列、数据行和数据表子查询可以返回更多的信息，但只能用在不要求必须使用单个值的上下文里。

子查询既可以与主查询相关——需要引用和依赖于主查询里的值，也可以与之不相关。

除了 SELECT 语句，子查询还可以用在其他语句里。不过，如果把子查询用在一条会改变数据表内容的语句（INSERT、REPLACE、DELETE、UPDATE、LOAD DATA）里，目前还必须遵守这样一条限制：子查询不允许引用正在被修改的数据表。

有些子查询可以被改写为联结查询。如果你编写的查询命令有可能被拿到一个版本较早的 MySQL 服务器上去运行，或者如果你想比较一下 MySQL 优化器对联结查询和子查询的优化效果，就应该多掌握一些子查询的改写技巧。

在接下来的几节里，我们将讨论几种可以用来测试子查询结果的操作，以及如何编写与主查询相关的子查询，如何把子查询改写为联结查询。

2.9.1　子查询与关系比较操作符

=、<>、>、>=、<和<=操作符用来进行相对值比较。它们可以和标量子查询配合使用：用子查询返回的值来构造一个检索条件，再由主查询根据该条件做进一步检索。比如说，如果你想查看 '2008-09-23' 那天的小测验成绩，可以用一个标量子查询来确定那次小测验的 event_id，再让主查询使用这个 event_id 去检索 score 数据表：

```
SELECT * FROM score
WHERE event_id =
(SELECT event_id FROM grade_event
   WHERE date = '2008-09-23' AND category = 'Q');
```

这种形式的语句有一个共同点：子查询的前面有一个值和一个相对比较操作符，子查询的这种用法要求它只返回一个值，也就是说，它必须是标量子查询；如果它返回了多个值，整条语句将以失败告终。有时候，为了确保子查询返回且只返回一个值，有必要用 LIMIT 1 对子查询的结果加以限制。

如果你遇到的问题可以通过在 WHERE 子句里使用某个聚合函数去解决，就应该考虑用标量子查询和相对比较操作符。比如说，如果你想知道 president 数据表里的哪位总统出生得最早，你可能会写出如下所示的语句：

```
SELECT * FROM president WHERE birth = MIN(birth);
```

可这个办法是行不通的——你不能在 WHERE 子句里使用聚合函数。WHERE 子句的用途是确定应该选取哪些数据行，但 MIN()的值只有在选取完数据行之后才能确定下来。不过，你可以像下面这样用一个子查询把最早的出生日期检索出来：

```
SELECT * FROM president
WHERE birth = (SELECT MIN(birth) FROM president);
```

其他的聚合函数可以用来解决类似的问题。下面这条语句使用了一个子查询来选取在某次考试高于平均分数的分数：

```
SELECT * FROM score WHERE event_id = 5
AND score > (SELECT AVG(score) FROM score WHERE event_id = 5);
```

如果子查询返回的是一个数据行，你可以用一个数据行构造器把一组值（一条临时创建的记录）与子查询的结果进行比较。下面这条语句将返回一个数据行，它可以告诉我们哪几位总统和 John Adams 出生在同一个城市和州：

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) =
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Adams' AND first_name = 'John');
+-----------+------------+-----------+-------+
| last_name | first_name | city      | state |
```

```
+-----------+-------------+-----------+-------+
| Adams     | John        | Braintree | MA    |
| Adams     | John Quincy | Braintree | MA    |
+-----------+-------------+-----------+-------+
```

你也可以使用 ROW(city, state)表示法，它等价于(city, state)。它们都可以用来生成一条临时记录。

2.9.2　IN和NOT IN子查询

如果你的子查询将返回多个数据行，你可以用 IN 和 NOT IN 操作符来构造主查询的检索条件。IN 和 NOT IN 操作符的用途是测试一个给定的比较值有没有出现在一个特定的集合里。只要主查询里的数据行与子查询所返回的任何一个数据行匹配，IN 操作符的比较结果就将是 true。如果主查询里的数据行与子查询所返回的所有数据行都不匹配，NOT IN 操作符的比较结果将是 true。下面两条语句分别使用了 IN 和 NOT IN 操作符来查找在 absence 数据表里有缺勤记录和没有缺勤记录（也就是全勤）的学生：

```
mysql> SELECT * FROM student
    -> WHERE student_id IN (SELECT student_id FROM absence);
+-------+-----+------------+
| name  | sex | student_id |
+-------+-----+------------+
| Kyle  | M   |          3 |
| Abby  | F   |          5 |
| Peter | M   |         10 |
| Will  | M   |         17 |
| Avery | F   |         20 |
+-------+-----+------------+
mysql> SELECT * FROM student
    -> WHERE student_id NOT IN (SELECT student_id FROM absence);
+-----------+-----+------------+
| name      | sex | student_id |
+-----------+-----+------------+
| Megan     | F   |          1 |
| Joseph    | M   |          2 |
| Katie     | F   |          4 |
| Nathan    | M   |          6 |
| Liesl     | F   |          7 |
...
```

IN 和 NOT IN 操作符还可以用在将返回多个数据列的子查询里。换句话说，你可以在数据表子查询里使用它们。此时，你需要使用一个数据行构造器来给出将与各数据列比较的比较值。

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) IN
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Roosevelt');
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Roosevelt | Theodore    | New York  | NY    |
| Roosevelt | Franklin D. | Hyde Park | NY    |
+-----------+-------------+-----------+-------+
```

IN 和 NOT IN 操作符其实是= ANY 和< > ALL 的同义词；详见下一节里的讨论。

2.9.3 ALL、ANY和SOME子查询

ALL 和 ANY 操作符的常见用法是结合一个相对比较操作符对一个数据列子查询的结果进行测试。它们测试比较值是否与子查询所返回的全部或一部分值匹配。比如说，如果比较值小于或等于子查询所返回的每一个值，<= ALL 将是 true；只要比较值小于或等于子查询所返回的任何一个值，<= ANY 将是 true。SOME 是 ANY 的一个同义词。

下面这条语句用来检索最早出生的总统，具体做法是选取出生日期小于或等于 president 数据表里的所有出生日期（只有最早的出生日期满足这一条件）的那个数据行：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth <= ALL (SELECT birth FROM president);
+------------+------------+------------+
| last_name  | first_name | birth      |
+------------+------------+------------+
| Washington | George     | 1732-02-22 |
+------------+------------+------------+
```

下面这条语句的用处就不大了，它将返回所有的数据行，因为对于每个日期，至少有一个日期（它本身）大于或等于它：

```
mysql> SELECT last_name, first_name, birth FROM president
    -> WHERE birth <= ANY (SELECT birth FROM president);
+------------+---------------+------------+
| last_name  | first_name    | birth      |
+------------+---------------+------------+
| Washington | George        | 1732-02-22 |
| Adams      | John          | 1735-10-30 |
| Jefferson  | Thomas        | 1743-04-13 |
| Madison    | James         | 1751-03-16 |
| Monroe     | James         | 1758-04-28 |
...
```

当 ALL、ANY 或 SOME 操作符与“=”比较操作符配合使用时，子查询可以是一个数据表子查询。此时，你需要使用一个数据行构造器来提供与子查询所返回的数据行进行比较的比较值。

```
mysql> SELECT last_name, first_name, city, state FROM president
    -> WHERE (city, state) = ANY
    -> (SELECT city, state FROM president
    -> WHERE last_name = 'Roosevelt');
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Roosevelt | Theodore    | New York  | NY    |
| Roosevelt | Franklin D. | Hyde Park | NY    |
+-----------+-------------+-----------+-------+
```

前一节里提到过，IN 和 NOT IN 操作符是= ANY 和< > ALL 的简写。也就是说，IN 操作符的含义是“等于子查询所返回的某个数据行”，NOT IN 操作符的含义是“不等于子查询所返回的任何数据行”。

2.9.4　EXISTS和NOT EXISTS子查询

EXISTS 和 NOT EXISTS 操作符只测试某个子查询是否返回了数据行：如果是，EXISTS 将是 true，NOT EXISTS 将是 false。下面两条语句演示了这两个操作符的用法。如果 absence 数据表是空白的，第一条语句将返回 0，第二条语句将返回 1：

```
SELECT EXISTS (SELECT * FROM absence);
SELECT NOT EXISTS (SELECT * FROM absence);
```

EXISTS 和 NOT EXISTS 操作符在与主查询相关的子查询里比较常见。详见 2.9.5 节里的讨论。

在使用 EXISTS 和 NOT EXISTS 操作符时，子查询通常使用“*”作为输出数据列清单。这两个操作符是根据子查询是否返回了数据行来判断真假的，不关心数据行所包含的内容是什么，所以没必要明确地列出数据列的名字。事实上，你可以在子查询的数据列选取清单里写出任何东西，但如果你想确保在子查询成功时返回一个 true 值的话，把它写成 SELECT 1 当然要比把它写成 SELECT *更保险。

2.9.5　与主查询相关的子查询

如下所示，子查询可以与主查询相关，也可以与之无关。

- 与主查询无关的子查询不引用主查询里的任何值。与主查询无关的子查询可以脱离主查询作为一条独立的查询命令去执行。比如说，下面这条语句里的子查询就是与主查询无关的，它只引用了数据表 t1，没有引用数据表 t2：

  ```
  SELECT j FROM t2 WHERE j IN (SELECT i FROM t1);
  ```

- 与主查询相关的子查询需要引用主查询里的值，所以必须依赖于主查询。因为这种联系，与主查询相关的子查询不能脱离主查询作为一条独立的查询命令去执行。比如说，下面这条语句里的子查询的作用是把t2数据表中数据列i的每一个值与数据表t1中数据列j的值相匹配：

  ```
  SELECT j FROM t2 WHERE (SELECT i FROM t1 WHERE i = j);
  ```

与主查询相关的子查询通常用在 EXISTS 和 NOT EXISTS 子查询里，这类子查询主要用来在某个数据表里查找在另一个数据表里有匹配数据行或没有匹配数据行的数据行。与主查询相关的子查询的工作情况是：把值从主查询传递到子查询，看它们是否满足在子查询里给出的条件。因此，如果数据列的名字有可能引起歧义的话（在不同的数据表里有同名的数据列），千万不要忘记使用数据表的名字来限定数据列。

下面的 EXISTS 子查询用来确定数据表之间的匹配——也就是在两个数据表里都有的值，而整个语句的用途是选取在 absence 数据表里至少有一条缺勤记录的学生：

```
SELECT student_id, name FROM student WHERE EXISTS
(SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

NOT EXISTS 操作符用来寻找不匹配的值（在一个数据表里有但在其他数据表里没有的值）。下面这条语句将把没有缺勤记录的学生查找出来：

```
SELECT student_id, name FROM student WHERE NOT EXISTS
(SELECT * FROM absence WHERE absence.student_id = student.student_id);
```

2.9.6　FROM子句中的子查询

子查询可以用在 FROM 子句里以生成一些值。在这种情况里，子查询的结果在行为上就像是一个

数据表。FROM 子句里的子查询可以参加关联操作，它的值可以在 WHERE 子句里被测试，等等。在使用这种子查询的时候，你必须提供一个数据表别名作为子查询的结果的名字：

```
mysql> SELECT * FROM (SELECT 1, 2) AS t1 INNER JOIN (SELECT 3, 4) AS t2;
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
| 1 | 2 | 3 | 4 |
+---+---+---+---+
```

2.9.7 把子查询改写为联结查询

有相当一部分使用了子查询的查询命令可以被改写为一个联结查询。有时候，联结查询要比子查询的执行效率更高，所以把子查询改写为联结查询是个不坏的主意。如果一条使用了子查询的 SELECT 语句需要很长时间执行完毕，就应该尝试把它改写为一个联结查询，看它是不是执行得更好。本节将介绍如何这样做。

1. 如何改写用来选取匹配值的子查询

下面这条示例语句包含一个子查询，它将从 score 数据表把考试（不包括小测验）成绩筛选出来：

```
SELECT * FROM score
WHERE event_id IN (SELECT event_id FROM grade_event WHERE category = 'T');
```

这条语句也可以写成不使用子查询的样子，只要把它转换为一个简单的联结操作就行了：

```
SELECT score.* FROM score INNER JOIN grade_event
ON score.event_id = grade_event.event_id WHERE grade_event.category = 'T';
```

再来看一个例子。下面这个查询只选取女生们的考试成绩：

```
SELECT * from score
WHERE student_id IN (SELECT student_id FROM student WHERE sex = 'F');
```

这条语句可以被改写为下面的联结查询：

```
SELECT score.* FROM score INNER JOIN student
ON score.student_id = student.student_id WHERE student.sex = 'F';
```

看出其中的规律了吗？这几个例子里的子查询语句都是如下所示的形式：

```
SELECT * FROM table1
WHERE column1 IN (SELECT column2a FROM table2 WHERE column2b = value);
```

这类查询可以被转换为如下所示的联结查询：

```
SELECT table1.* FROM table1 INNER JOIN table2
ON table1.column1 = table2.column2a WHERE table2.column2b = value;
```

在某些场合，子查询和关联查询可能返回不同的结果。这发生在 table2 包含多个 column2a 的实例的时候。子查询只为每个 column2a 值生成一个实例，联结操作会把它们都生成出来并导致在其输出里出现重复的数据行。如果你想防止这种重复，在编写联结查询命令时就要用 SELECT DISTINCT 来代替 SELECT。

2. 如何改写用来选取非匹配（缺失）值的子查询

子查询语句的另一种常见用法是检索在一个数据表里有、在另一个数据表里没有的值。前面看到过，与“哪些值没有出现”有关的问题通常都可以用 LEFT JOIN 来解决。下面是一个我们曾经见过的

子查询，它用来测试哪些学生没有出现在 absence 数据表里（查找全勤的学生）：

```
SELECT * FROM student
WHERE student_id NOT IN (SELECT student_id FROM absence);
```

这条语句可以被改写为如下所示的 LEFT JOIN 查询命令：

```
SELECT student.*
FROM student LEFT JOIN absence ON student.student_id = absence.student_id
WHERE absence.student_id IS NULL;
```

一般来说，如果子查询语句有如下所示的形式：

```
SELECT * FROM table1
WHERE column1 NOT IN (SELECT column2 FROM table2);
```

就可以把它改写为下面这样的联结查询：

```
SELECT table1.*
FROM table1 LEFT JOIN table2 ON table1.column1 = table2.column2
WHERE table2.column2 IS NULL;
```

注意，这里需要假设 table2.column2 被定义成 NOT NULL。

与 LEFT JOIN 相比，子查询具有比较直观和容易理解的优点。绝大多数人都可以毫无困难地理解“没被包含在……里面”的含义，它不是数据库编程技术带来的新概念。“左联结”这个概念就不同了，就算是数据库方面的专业人士也不见得都能把它解释透彻。

2.10 用 UNION 语句进行多数据表检索

如果想把多个查询的结果合并在一起创建一个结果集，可以使用 UNION 语句来做这件事。本节中的例子假设你有 t1、t2 和 t3 这 3 个数据表，如下所示：

```
mysql> SELECT * FROM t1;
+------+-------+
| i    | c     |
+------+-------+
|    1 | red   |
|    2 | blue  |
|    3 | green |
+------+-------+
mysql> SELECT * FROM t2;
+------+------+
| i    | c    |
+------+------+
|   -1 | tan  |
|    1 | red  |
+------+------+
mysql> SELECT * FROM t3;
+------------+------+
| d          | i    |
+------------+------+
| 1904-01-01 |  100 |
| 2004-01-01 |  200 |
| 2004-01-01 |  200 |
+------------+------+
```

数据表 t1 和 t2 有整数和字符数据列，t3 有日期和整数数据列。为了编写一条 UNION 语句把多个检索合并在一起，需要先写出几条 SELECT 语句（它们检索出来的数据列个数相同），然后把关键字 UNION 放到它们之间。比如说，下面这条语句将选取各数据表的整数数据列并把它们合并在一起：


```
mysql> SELECT i FROM t1 UNION SELECT i FROM t2 UNION SELECT i FROM t3;
+------+
| i    |
+------+
|    1 |
|    2 |
|    3 |
|   -1 |
|  100 |
|  200 |
+------+
```

UNION 语句有以下特性。

数据列的名字和数据类型。UNION 结果集里的数据列名字来自第一个 SELECT 语句里的数据列的名字。UNION 中的第二个和再后面的 SLEECT 语句必须选取个数相同的数据列，但各有关数据列不必有同样的名字或数据类型。（一般来说，同一条 UNION 语句里的各有关数据列会是相同的数据类型，但这并不是一项要求。如果它们不一样，MySQL 会进行必要的类型转换。）数据列是根据位置而不是根据名字进行匹配的，这也正是下面两条语句会返回不同结果的原因，虽然它们从两个数据表选取的是同样的值：

```
mysql> SELECT i, c FROM t1 UNION SELECT i, d FROM t3;
+------+------------+
| i    | c          |
+------+------------+
|    1 | red        |
|    2 | blue       |
|    3 | green      |
|  100 | 1904-01-01 |
|  200 | 2004-01-01 |
+------+------------+
mysql> SELECT i, c FROM t1 UNION SELECT d, i FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
+------------+-------+
```

在每条语句里，结果中的每个数据列的类型是根据被选取的值而确定的。在第一条语句里，我们为第二个数据列选取的是字符串的日期，结果是一个字符串数据列。在第二条语句里，为第一个数据列选取的是整数和日期，为第二个数据列选取的是字符串和整数。在这两种情况里，结果都是一个字符串数据列。

重复数据行的处理。在默认的情况下，UNION 将从结果集里剔除重复的数据行：

```
mysql> SELECT * FROM t1 UNION SELECT * FROM t2 UNION SELECT * FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| -1         | tan   |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
+------------+-------+
```

t1 和 t2 都有一个包含着 1 和'red'值的数据行，但输出结果里只有一个这样的数据行。此外，t3 有两个包含'2004-01-01'和 200 的数据行，其中之一被剔除了。

UNION DISTINCT 是 UNION 的同义词，它们都只保留不重复的数据行。

如果你想保留重复的数据行，需要把每个 UNION 都改为 UNION ALL。

```
mysql> SELECT * FROM t1 UNION ALL SELECT * FROM t2 UNION ALL SELECT * FROM t3;
+------------+-------+
| i          | c     |
+------------+-------+
| 1          | red   |
| 2          | blue  |
| 3          | green |
| -1         | tan   |
| 1          | red   |
| 1904-01-01 | 100   |
| 2004-01-01 | 200   |
| 2004-01-01 | 200   |
+------------+-------+
```

如果把 UNION（或 UNION DISTINCT）和 UNION ALL 混杂在一起使用，每个 UNION（或 UNION DISTINCT）操作将优先于它左边的任何 UNION ALL 操作。

ORDER BY 和 LIMIT 处理。如果你想将 UNION 结果作为一个整体进行排序，需要用括号把每一个 SELECT 语句括起来并在最后一个 SELECT 语句的后面加上一个 ORDER BY 子句。注意，因为 UNION 的结果数据列的名字来自第一个 SELECT 语句，所以你在 ORDER BY 子句里必须引用那些名字而不是引用来自最后一个 SELECT 语句的数据列名字：

```
mysql> (SELECT i, c FROM t1) UNION (SELECT i, d FROM t3)
    -> ORDER BY c;
+------+------------+
| i    | c          |
+------+------------+
|  100 | 1904-01-01 |
|  200 | 2004-01-01 |
|    2 | blue       |
|    3 | green      |
|    1 | red        |
+------+------------+
```

如果某个排序数据列有别名，位于 UNION 语句末尾的 ORDER BY 子句必须引用那个别名。此外，ORDER BY 不能引用数据表的名字。如果为了排序而需要以 *table_name.col_name* 的形式给出一个来

自第一个 SELECT 语句的数据列名字，必须给那个数据列起一个别名并在 ORDER BY 子句里引用那个别名。

类似地，如果你想限制 UNION 语句所输出的数据行的个数，可以在语句末尾加上一个 LIMIT 子句。

```
mysql> (SELECT * FROM t1) UNION (SELECT * FROM t2) UNION (SELECT * FROM t3)
    -> LIMIT 2;
+------+-------------+
| i    | c           |
+------+-------------+
| 1    | red         |
| 2    | blue        |
+------+-------------+
```

在 UNION 语句中，ORDER BY 和 LIMIT 还可以用在被括号括起来的各条 SELECT“子句”里。此时，它们将只作用于那条 SELECT 语句：

```
mysql> (SELECT * FROM t1 ORDER BY i LIMIT 2)
    -> UNION (SELECT * FROM t2 ORDER BY i LIMIT 1)
    -> UNION (SELECT * FROM t3 ORDER BY d LIMIT 2);
+------------+-------------+
| i          | c           |
+------------+-------------+
| 1          | red         |
| 2          | blue        |
| -1         | tan         |
| 1904-01-01 | 100         |
| 2004-01-01 | 200         |
+------------+-------------+
```

在用括号括起来的各个 SELECT 语句里的 ORDER BY 只能在 LIMIT 也出现时才能使用，以确定 LIMIT 将作用于哪些数据行。此时，它只影响那条 SELECT 语句的结果，不影响最终的 UNION 结果里的数据行的先后顺序。

如果你想在一组结构相同的 MyISAM 数据表上运行 UNION 类型的查询，建议你创建一个 MERGE 数据表并查询，因为编写针对单个 MERG 数据表的查询命令一般都要比编写相应的 UNION 语句容易一些。对 MERGE 数据表进行查询类似于用一条 UNOIN 语句从它的各成员数据表里把相应的数据列选取出来。具体地说，MERGE 数据表上的 SELECT 语句相当于 UNION ALL（不剔除重复的数据行），SELECT DISTINCT 相当于 UNION 或 UNION DISTINCT（剔除重复的数据行）。

2.11 使用视图

视图是一种虚拟的数据表，它们的行为和数据表一样，但并不真正包含数据。它们是用底层（真正的）数据表或其他视图定义出来的“假”数据表，用来提供查看数据表数据的另一种方法，这通常可以简化应用程序。

本节重点介绍视图的一些应用。这里没有讨论 DEFINER 子句，这个子句是存储程序和视图都使用的，它可以用来从信息安防的角度对视图数据的访问情况进行控制。关于 DEFINER 子句的详细讨论参见 4.5 节。

如果要选取某给定数据表的数据列的一个子集，把它定义为一个简单的视图是最方便的做法。比

如说，假设你经常需要从 president 数据表选取 last_name、first_name、city 和 state 等几个数据列，但不想每次都必须写出所有这些数据列，如下所示：

```
SELECT last_name, first_name, city, state FROM president;
```

你也不想使用 SELECT *，这虽然简单，但用*检索出来的数据列不都是你想要的。解决这个矛盾的办法是定义一个视图，让它只包括你想要的数据列：

```
CREATE VIEW vpres AS
SELECT last_name, first_name, city, state FROM president;
```

这个视图就像一个“窗口”，从中只能看到你想看的数据列。这意味着你可以在这个视图上使用 SELECT *，而你看到的将是你在视图定义里给出的那些数据列：

```
mysql> SELECT * FROM vpres;
+------------+---------------+---------------------+-------+
| last_name  | first_name    | city                | state |
+------------+---------------+---------------------+-------+
| Washington | George        | Wakefield           | VA    |
| Adams      | John          | Braintree           | MA    |
| Jefferson  | Thomas        | Albemarle County    | VA    |
| Madison    | James         | Port Conway         | VA    |
| Monroe     | James         | Westmoreland County | VA    |
...
```

如果你在查询某个视图时还使用了一个 WHERE 子句，MySQL 将在执行该查询时把它添加到那个视图的定义上以进一步限制其检索结果：

```
mysql> SELECT * FROM vpres WHERE last_name = 'Adams';
+-----------+-------------+-----------+-------+
| last_name | first_name  | city      | state |
+-----------+-------------+-----------+-------+
| Adams     | John        | Braintree | MA    |
| Adams     | John Quincy | Braintree | MA    |
+-----------+-------------+-----------+-------+
```

在查询视图时还可以使用 ORDER BY、LIMIT 等子句，其效果与查询一个真正的数据表时的情况一样。

在使用视图时，你只能引用在该视图的定义里列出的数据列。也就是说，如果底层数据表里的某个数据列没在视图的定义里，你在使用视图的时候就不能引用它：

```
mysql> SELECT * FROM vpres WHERE suffix <> '';
ERROR 1054 (42S22): Unknown column 'suffix' in 'where clause'
```

在默认的情况下，视图里的数据列的名字与 SELECT 语句里列出的输出数据列相同。如果你想明确地改用另外的数据列名字，需要在定义视图时在视图名字的后面用括号列出那些新名字：

```
mysql> CREATE VIEW vpres2 (ln, fn) AS
    -> SELECT last_name, first_name FROM president;
```

此后，当你使用这个视图时，必须使用在括号里给出的数据列名字，而非 SELECT 语句里的名字：

```
mysql> SELECT last_name, first_name FROM vpres2;
ERROR 1054 (42S22) at line 1: Unknown column 'last_name' in 'field list'
mysql> SELECT ln, fn FROM vpres2;
```

```
+------------+---------------+
| ln         | fn            |
+------------+---------------+
| Washington | George        |
| Adams      | John          |
| Jefferson  | Thomas        |
| Madison    | James         |
| Monroe     | James         |
...
```

视图可以用来自动完成必要数学运算。1.4.9 节中的第 6 小节编写了一条语句来确定各位总统去世时的年龄，我们可以把同样的数学运算放到一个视图定义里进行：

```
mysql> CREATE VIEW pres_age AS
    -> SELECT last_name, first_name, birth, death,
    -> TIMESTAMPDIFF(YEAR, birth, death) AS age
    -> FROM president;
```

这个视图包含一个 age 数据列，它被定义成一个运算，从这个视图选取该数据列将检索出这个运算的结果：

```
mysql> SELECT * FROM pres_age;
+------------+---------------+------------+------------+------+
| last_name  | first_name    | birth      | death      | age  |
+------------+---------------+------------+------------+------+
| Washington | George        | 1732-02-22 | 1799-12-14 |   67 |
| Adams      | John          | 1735-10-30 | 1826-07-04 |   90 |
| Jefferson  | Thomas        | 1743-04-13 | 1826-07-04 |   83 |
| Madison    | James         | 1751-03-16 | 1836-06-28 |   85 |
| Monroe     | James         | 1758-04-28 | 1831-07-04 |   73 |
...
```

通过把年龄计算工作放到视图定义里完成，我们就用不着再在查询年龄值时写出那个公式了。有关的细节都隐藏在了视图里。

同一个视图可以涉及多个数据表，这使得联结查询的编写和运行变得更容易。下面定义的视图对 score、student 和 grade_event 数据表进行了联结查询：

```
mysql> CREATE VIEW vstudent AS
    -> SELECT student.student_id, name, date, score, category
    -> FROM grade_event INNER JOIN score INNER JOIN student
    -> ON grade_event.event_id = score.event_id
    -> AND score.student_id = student.student_id;
```

当你从这个视图选取数据时，MySQL 将执行相应的联结查询并从多个数据表返回信息：

```
mysql> SELECT * FROM vstudent;
+------------+-----------+------------+-------+----------+
| student_id | name      | date       | score | category |
+------------+-----------+------------+-------+----------+
|          1 | Megan     | 2008-09-03 |    20 | Q        |
|          3 | Kyle      | 2008-09-03 |    20 | Q        |
|          4 | Katie     | 2008-09-03 |    18 | Q        |
|          5 | Abby      | 2008-09-03 |    13 | Q        |
|          6 | Nathan    | 2008-09-03 |    18 | Q        |
|          7 | Liesl     | 2008-09-03 |    14 | Q        |
```

```
|          8 | Ian       | 2008-09-03 |    14 | Q        |
...
```

这个视图可以让我们轻而易举地根据名字检索出某个学生的考试成绩：

```
mysql> SELECT * FROM vstudent WHERE name = 'emily';
+------------+-------+------------+-------+----------+
| student_id | name  | date       | score | category |
+------------+-------+------------+-------+----------+
|         31 | Emily | 2008-09-03 |    11 | Q        |
|         31 | Emily | 2008-09-06 |    19 | Q        |
|         31 | Emily | 2008-09-09 |    81 | T        |
|         31 | Emily | 2008-09-16 |    19 | Q        |
|         31 | Emily | 2008-09-23 |     9 | Q        |
|         31 | Emily | 2008-10-01 |    76 | T        |
+------------+-------+------------+-------+----------+
```

有些视图是可更新的，这意味着你可以通过对视图进行操作而在其底层数据表里插入、更新或删除数据行。下面是一个简单的例子：

```
mysql> CREATE TABLE t (i INT);
mysql> INSERT INTO t (i) VALUES(1),(2),(3);
mysql> CREATE VIEW v AS SELECT i FROM t;
mysql> SELECT i FROM v;
+------+
| i    |
+------+
|    1 |
|    2 |
|    3 |
+------+
mysql> INSERT INTO v (i) VALUES(4);
mysql> DELETE FROM v WHERE i < 3;
mysql> SELECT i FROM v;
+------+
| i    |
+------+
|    3 |
|    4 |
+------+
mysql> UPDATE v SET i = i + 1;
mysql> SELECT i FROM v;
+------+
| i    |
+------+
|    4 |
|    5 |
+------+
```

要想让一个视图是可更新的，它必须直接映射到一个数据表上，它选取的数据列只能是数据表里数据列的简单引用（不能是表达式），视图里的单行操作必须对应于对其底层数据表里的一个单行操作。比如说，如果某个视图里有一个“汇总”数据列是用一个聚合函数计算出来的，这个视图里的每个数据行都将涉及其底层数据表里的多个数据行。这样的视图是不可更新的，因为你无法告诉MySQL应该更新其底层数据表里的哪一个数据行。

2.12 涉及多个数据表的删除和更新操作

有时候，我们可以根据某个数据表里数据行是否在另一个数据表里有匹配来删除它们，这很有用。类似地，用一个数据表里的数据行的内容去更新另一个数据表也很有用。本节讨论如何完成涉及多个数据表的 DELETE 和 UPDATE 操作。联结概念在用来完成这些操作的语句里扮演着极其重要的角色，这要求你对前面 2.8 节里讨论的内容有透彻的理解。

对于只涉及单个数据表的 DELETE 和 UPDATE 操作，被引用的数据列都来自同一个数据表，不需要使用数据表的名字对数据列的名字进行限定。比如说，如果要从数据表 t 里把 id 值大于 100 的数据行全部删掉，可以编写如下所示的语句：

```
DELETE FROM t WHERE id > 100;
```

如果需要根据某给定数据表里数据行与另一个数据表里的数据行之间的关系（而不是根据其自身的属性）来删除它们，你该怎么办？比如说，如果要从数据表 t 里把其 id 值可以在另一个数据表 t2 里找到的数据行删掉，该怎么办？

在编写一个涉及多个数据表的 DELETE 语句时，要把所涉及的数据表在 FROM 子句里全部列出来并把用来匹配各有关数据行（它们来自多个数据表）的检索条件写在 WHERE 子句里。下面这条语句将从数据表 t1 里把其 id 值可以在另一个数据表 t2 里找到的数据行全部删掉：

```
DELETE t1 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

请注意，如果某个数据列的名字出现在了多个数据表里，就有可能导致歧义问题，就需要用数据表的名字对它加以限定。

DELETE 语句有一种语法可以让我们一次删除多个数据表里的数据行。如果你想从两个数据表里把 id 值相匹配的数据行都删掉，必须在 DELETE 关键字的后面写出两个数据表的名字：

```
DELETE t1, t2 FROM t1 INNER JOIN t2 ON t1.id = t2.id;
```

如果你想删除的是不匹配的数据行，又该怎么办？在涉及多个数据表的 DELETE 语句里可以使用允许用在 SELECT 语句里的任何一种联结操作，所以我们可以按照编写一条从多个数据表选取不匹配数据行的 SELECT 语句的思路去思考。这通常都会归结到是选用 LEFT JOIN 还是选用 RIGHT JOIN 上。比如说，如果要从数据表 t1 里把在数据表 t2 里没有匹配的数据行找出来，你应该会写出一条如下所示的 SELECT 语句：

```
SELECT t1.* FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

同样，从数据表 t1 找出并删除那些数据行的 DELETE 语句也要用到一个 LEFT JOIN 操作：

```
DELETE t1 FROM t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

MySQL 还支持另一种涉及多个数据表的 DELETE 语法。这种语法使用一个 FROM 子句来列出将从中删除有关数据行的数据表，使用一个 USING 子句来联结各有关数据表以确定哪些数据行需要被删除。前面那几条涉及多个数据表的 DELETE 语句可以用这种语法改写为如下所示的样子：

```
DELETE FROM t1 USING t1 INNER JOIN t2 ON t1.id = t2.id;
DELETE FROM t1, t2 USING t1 INNER JOIN t2 ON t1.id = t2.id;
DELETE FROM t1 USING t1 LEFT JOIN t2 ON t1.id = t2.id WHERE t2.id IS NULL;
```

编写涉及多个数据表的 UPDATE 语句的基本步骤与编写涉及多个数据表的 DELETE 语句的很相似。同样需要列出所涉及的全部数据表，同样需要用数据表的名字对数据列的名字进行必要的限定。我们

来看一个例子。假设 2008 年 9 月 23 日的测试里有一个问题是所有学生都没有答对的，而你后来发现这是因为你的标准答案有错误。于是，你决定给每位学生的考试成绩加上一分。下面这条涉及多个数据表的 UPDATE 语句可以完成这个工作：

```
UPDATE score, grade_event SET score.score = score.score + 1
WHERE score.event_id = grade_event.event_id
AND grade_event.date ='2008-09-23' AND grade_event.category = 'Q';
```

具体到这个问题，你也可以用一个基于子查询的单数据表更新操作来达到同样的目的：

```
UPDATE score SET score = score + 1
WHERE event_id = (SELECT event_id FROM grade_event
WHERE date = '2008-09-23' AND category = 'Q');
```

但其他类型的更新操作能不能用子查询来完成就不一定了。比如说，假设你不仅需要根据另一个数据表的内容来确定应该刷新某给定数据表里的哪些数据行，还需要把另一个数据表的数据列值复制到这个数据表里。下面这条语句将把符合条件的 t1.a 复制到 t2.a，而需要满足的条件是数据行有匹配的 id 数据列值：

```
UPDATE t1, t2 SET t2.a = t1.a WHERE t2.id = t1.id;
```

如果是对 InnoDB 数据表进行多数据表删除和刷新操作，你不必非得使用刚才介绍的语法。更好的办法是在数据表之间建立一个外键关系并给它加上 ON DELETE CASCADE 或 ON UPDATE CASCADE 约束条件。详见 2.14 节。

2.13　事务处理

事务（transaction）是作为一个不可分割的逻辑单元而被执行的一组 SQL 语句，如有必要，它们的执行效果可以被撤销。并非所有的语句每次都能执行成功，有些语句还会对数据产生永久性的影响。事务处理是通过提交（commit）和回滚（rollback）功能实现的。如果某个事务里的所有语句都执行成功了，提交该事务将把那些语句的执行效果永久性地记录到数据库里。如果在事务过程中发生错误，回滚该事务将把发生错误之前已经执行的语句全部取消，数据库将恢复到开始这次事务之前的状态。

提交和回滚机制使我们能够确保尚未全部完成的操作不会影响到数据库，不会让新旧数据混杂在一起让数据库呈不稳定状态。财务转账是一个典型的事务处理例子，即把钱从一个账户转到另一个账户。假设 Bill 给 Bob 开了一张 100 美元的支票，Bill 拿着这张支票去取钱。Bill 的账户应该减少 100 美元，Bob 的账户应该增加 100 美元：

```
UPDATE account SET balance = balance - 100 WHERE name = 'Bill';
UPDATE account SET balance = balance + 100 WHERE name = 'Bob';
```

可是，万一银行的计算机系统在这两条语句正在执行时发生了崩溃，整个操作将不完整。根据先执行的是哪一条语句，Bill 的账户可能少了 100 美元而 Bob 的账户金额没增加，或者 Bob 的账户多了 100 美元而 Bill 的账户金额没减少。这两种情况都不正确。如果没有使用事务机制，你将不得不以手动方式分析你的日志以查明崩溃发生时都有哪些操作正在进行，应该以及如何撤销或继续完成哪些操作，等等。事务机制提供的回滚操作可以让你正确地处理好这些问题，把发生错误之前已经执行完的语句的效果撤销掉。（作为善后工作的一部分，你还需要确定哪些事务需要再次执行，但至少你不必担心那些未能全部完成的事务会损害数据库的完整性了。）

事务的另一种用途是确保某个操作所涉及的数据行在你正在使用它们时不会被其他客户修改。

MySQL 在执行每一条 SQL 语句时都会自动地对该语句所涉及的资源进行锁定以避免各语句之间相互干扰。但这仍不足以保证每一个数据库操作总是能够得到预期的结果。要知道，有些数据库操作需要多条语句才能完成，而在此期间，不同的客户就有可能相互干扰。通过把多条语句定义为一个执行单元，事务机制可以防止在多客户环境里可能发生的并发问题。

事务机制的特性通常被概括为“ACID 原则”。ACID 是 Atomic（原子性）、Consistent（稳定性）、Isolated（孤立性）和 Durable（可靠性）的首字母缩写，它们分别代表事务机制应该具备的一个属性。

- **原子性**。构成一个事务的所有语句应该是一个独立的逻辑单元，要么全部执行成功，要么一个都不成功。你不能只执行它们当中的一部分。
- **稳定性**。数据库在事务开始执行之前和事务执行完毕之后都必须是稳定的。换句话说，事务不应该把你的数据库弄得一团糟。
- **隔离性**。事务不应该相互影响。
- **可靠性**。如果事务执行成功，它的影响将被永久性地记录到数据库里。

事务处理为数据库操作的结果提供了强有力的保证，但这需要以增加 CPU、内存和硬盘空间等方面的开销为代价。MySQL 提供了几种具备事务安全性的存储引擎（如 InnoDB 和 Falcon）和一些不具备事务安全性的存储引擎（如 MyISAM 和 MEMORY）。有些应用程序需要通过事务来实现，另外一些则不然，你可以根据具体情况挑选最适合的。一般来说，与金融有关的操作应该以事务方式完成，这是因为财务数据的完整性要比额外增加的开销成本更重要。从另一方面看，对于一个负责把 Web 页面的访问情况记入数据库表的应用程序来说，在主机崩溃时损失一些数据行应该是可以忍受的，你可以选用一种非事务存储引擎以避免事务处理所要求的额外开销。

2.13.1 利用事务来保证语句的安全执行

要想使用事务，就必须选用一种支持事务处理的存储引擎，如 InnoDB 或 Falcon。MyISAM 和 MEMORY 等其他存储引擎帮不上这个忙。如果你拿不准 MySQL 服务器是否支持任何事务存储引擎，请参见 2.6.1 节中的第 1 小节。

在默认的情况下，MySQL 从自动提交（autocommit）模式运行，这种模式会在每条语句执行完毕后把它作出的修改立刻提交给数据库并使之永久化。事实上，这相当于把每一条语句都隐含地当做一个事务来执行。如果你想明确地执行事务，需要禁用自动提交模式并告诉 MySQL 你想让它在何时提交或回滚有关的修改。

执行事务的常用办法是发出一条 `START TRANSACTION`（或 `BEGIN`）语句挂起自动提交模式，然后执行构成本次事务的各条语句，最后用一条 `COMMIT` 语句结束事务并把它们作出的修改永久性地记入数据库。万一在事务过程中发生错误，用一条 `ROLLBACK` 语句撤销事务并把数据库恢复到事务开始之前的状态。`START TRANSACTION` 语句“挂起”自动提交模式的含义是：在事务被提交或回滚之后，该模式将恢复到开始本次事务的 `START TRANSACTION` 语句被执行之前的状态。（如果自动提交模式原来是激活的，结束事务将让你回到自动提交模式；如果它原来是禁用的，结束当前事务将开始下一个事务。）

下面的例子演示了这个套路。首先，创建一个数据表供演示使用：

```
mysql> CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE = InnoDB;
```

这个语句将创建一个 InnoDB 数据表，但你完全可以根据个人喜好选用一种不同的事务存储引擎。

接下来，用 START TRANSACTION 语句开始一次事务，往数据表里添加一些数据行，提交本次事务，然后看看数据表变成了什么样子：

```
mysql> START TRANSACTION;
mysql> INSERT INTO t SET name = 'William';
mysql> INSERT INTO t SET name = 'Wallace';
mysql> COMMIT;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

你可以看到一些数据行已被记录到了数据表里。如果你另外启动一个 mysql 程序的实例并在插入之后、但提交之前选取数据表 t 的内容的话，你将看不到那些数据行。在第一个 mysql 进程发出 COMMIT 语句之前，那些数据行对第二个 mysql 进程来说是不可见的。

如果子事务过程中发生了一个错误，你可以用 ROLLBACK 语句把它删除。仍以数据表 t 为例，你可以通过发出下面这些语句看到这种情况：

```
mysql> START TRANSACTION;
mysql> INSERT INTO t SET name = 'Gromit';
mysql> INSERT INTO t SET name = 'Wallace';
ERROR 1062 (23000): Duplicate entry 'Wallace' for key 1
mysql> ROLLBACK;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

第二条 INSERT 语句试图把一个其 name 值与一个现有的数据行重复的数据行插入到数据表里。因为 name 数据列有一个 UNIQUE 索引，所以这条语句将执行失败。在发出 ROLLBACK 语句之后，这个数据表只包含在这次失败事务之前被插入的两个数据行。准确地说，在这次事务里，在发生错误之前已经执行的 INSERT 语句被撤销了，它们的影响没有被记录到数据表里。

如果在事务过程中发出一条 START TRANSACTION 语句，它将隐含地提交当前事务，然后开始一个新的事务。

执行事务的另一个办法是利用 SET 语句直接改变自动提交模式的状态：

```
SET autocommit = 0;
SET autocommit = 1;
```

把 autocommit 变量设置为零将禁用自动提交模式，其效果是随后的任何语句都将成为当前事务的一部分，直到你发出一条 COMMIT 或 ROLLBACK 语句来提交或撤销它为止。如果使用这个办法，自动提交模式的状态将一直保持下去直到你把它设置回原来的状态，所以结束一个事务将开始下一个事务。你也可以通过重新激活自动提交模式的办法来提交一个事务。

我们来看看这个办法的工作情况。首先，创建一个和前面那个例子一样的数据表：

```
mysql> DROP TABLE t;
mysql> CREATE TABLE t (name CHAR(20), UNIQUE (name)) ENGINE = InnoDB;
```

接下来，禁用自动提交模式，插入一些数据行，提交本次事务：

2

```
mysql> SET autocommit = 0;
mysql> INSERT INTO t SET name = 'William';
mysql> INSERT INTO t SET name = 'Wallace';
mysql> COMMIT;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

现在，有两个数据行被插入了数据表，但自动提交模式仍处于被禁用状态。如果你继续发出一些语句，它们将成为一个新事务的组成部分，它们的提交或撤销与第一个事务不会有任何关系。下面这些语句可以证明自动提交模式仍处于被禁用状态，并且 ROLLBACK 将撤销尚未被提交的语句：

```
mysql> INSERT INTO t SET name = 'Gromit';
mysql> INSERT INTO t SET name = 'Wallace';
ERROR 1062 (23000): Duplicate entry 'Wallace' for key 1
mysql> ROLLBACK;
mysql> SELECT * FROM t;
+---------+
| name    |
+---------+
| Wallace |
| William |
+---------+
```

要想重新激活自动提交模式，使用下面这条语句即可：

```
mysql> SET autocommit = 1;
```

正如刚才描述的那样，发出一条 COMMIT 或 ROLLBACK 语句将结束当前事务，先禁用自动提交模式、再重新激活它也将结束当前事务。在其他坏境下，事务也会结束。SET autocommit、STRAT TRANSACTION、BEGIN、COMMIT 和 ROLLBACK 语句会明确地对事务产生影响，除它们以外，还有一些语句会对事务产生隐式影响，因为它们不能成为事务的一部分。一般来说，用来创建、改变或删除数据库或其中的对象的 DLL（Data Definition Language，数据定义语言）语句以及与锁定有关的语句都不能成为事务的一部分。比如说，如果你在事务过程中发出了下面这些语句之一，服务器将在执行该语句之前先提交当前事务：

```
ALTER TABLE
CREATE INDEX
DROP DATABASE
DROP INDEX
DROP TABLE
LOCK TABLES
RENAME TABLE
SET autocommit = 1 (if not already set to 1)
```

```
TRUNCATE TABLE
UNLOCK TABLES (if tables currently are locked)
```

如果你想知道你正在使用的 MySQL 版本都有哪些语句会隐含地提交当前事务，请查阅《MySQL 参考手册》。

事务提交前，客户连接的正常结束或意外中断也将导致事务结束。此时，服务器会自动回滚该客户正提交的所有事务。

如果客户程序在与服务器连接意外断开后再自动重建连接，新建的连接将被重置为激活自动提交模式的默认状态。

有许多实际问题需要依靠事务才能确保它们被正确地解决。比如说，假设你有一个用来记录学生成绩的应用项目，其中有一个 score 数据表，你发现有两名学生的考试分数输入错了，需要交换一下。输入有误的考试成绩如下所示：

```
mysql> SELECT * FROM score WHERE event_id = 5 AND student_id IN (8,9);
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          8 |        5 |    18 |
|          9 |        5 |    13 |
+------------+----------+-------+
```

要纠正这个错误，应把 8 号学生的成绩改成 13，把 9 号学生的成绩改为 18。只需两条简单的语句就可以完成：

```
UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
```

可是，你必须保证这两条语句是作为一个不可分割的逻辑单元而执行成功的。这是一个需要使用事务来解决的典型问题。下面是用 START TRANSACTION 语句开始一个事务来解决这个问题的具体做法：

```
mysql> START TRANSACTION;
mysql> UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
mysql> UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
mysql> COMMIT;
```

下面通过直接设置自动提交模式来完成同样的事情：

```
mysql> SET autocommit = 0;
mysql> UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
mysql> UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
mysql> COMMIT;
mysql> SET autocommit = 1;
```

两种办法都可以保证那两名学生的考试成绩被正确地交换过来：

```
mysql> SELECT * FROM score WHERE event_id = 5 AND student_id IN (8,9);
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          8 |        5 |    13 |
|          9 |        5 |    18 |
+------------+----------+-------+
```

2.13.2　使用事务保存点

MySQL 使你能够对一个事务进行部分回滚。这需要你在事务过程中使用 SAVEPOINT 语句设置一些称为保存点（savepoint）的标记。在后续的事务里，如果你想回滚到某个特定的保存点，在 ROLLBACK 语句里给出改保存点的名字就可以了。下面的语句演示了这个过程：

```
mysql> CREATE TABLE t (i INT) ENGINE = InnoDB;
mysql> START TRANSACTION;
mysql> INSERT INTO t VALUES(1);
mysql> SAVEPOINT my_savepoint;
mysql> INSERT INTO t VALUES(2);
mysql> ROLLBACK TO SAVEPOINT my_savepoint;
mysql> INSERT INTO t VALUES(3);
mysql> COMMIT;
mysql> SELECT * FROM t;
+------+
| i    |
+------+
|    1 |
|    3 |
+------+
```

在执行完这些语句之后，数据表里只有第一条和第三条 INSERT 语句插入的数据行，没有第二条 INSERT 语句插入的数据行，它被那条含义是“回滚到 my_savepoint 保存点”的 ROLLBACK 语句给取消了。

2.13.3　事务的隔离性

因为 MySQL 是一个多用户数据库系统，所以不同的客户可能会在同一时间试图访问同一个数据表。诸如 MyISAM 之类的存储引擎使用了数据表级的锁定机制来保证不同的客户不能同时修改同一个数据表，但这种做法在更新量比较大的系统上会导致并发性能的下降。InnoDB 存储引擎采用了另一种策略，它使用了数据行级的锁定机制为客户对数据表的访问提供了更细致的控制：在某个客户修改某个数据行的同时，另一个客户可以读取和修改同一个数据表里的另一个数据行。如果有两个客户想同时修改某个数据行，先锁定该数据行的那个客户将可以先修改它。这比数据表级的锁定机制提供了更好的并发性能。不过，这里还有一个问题：一个客户的事务在何时才能看到另一个客户的事务作出的修改。

InnoDB 存储引擎实现的事务隔离级别机制能够让客户控制他们想看到其他事务作的哪些修改。它提供了多种不同的隔离级别以允许或预防在多个事务同时运行时可能发生的各种各样的问题，如下所示。

- 脏读（dirty read）。指某个事务所作出的修改在它尚未被提交时就可以被其他事务看到。其他事务会认为数据行已经被修改了，但对数据行作出修改的那个事务还有可能会被回滚，这将导致数据库里的数据发生混乱。
- 不可重复读取（nonrepeatable read）。指同一个事务使用同一条 SELECT 语句每次读取到的结果不一样。比如说，如果有一个事务执行了两次同一个 SELECT 语句，但另一个事务在这条 SELECT 语句的两次执行之间修改了一些数据行，就会发生这种问题。
- 幻影数据行（phantom row）。指某个事务突然看到了一个它以前没有见过数据行。比如说，如

果某个事务刚执行完一条 SELECT 语句就有另一个事务插入了一个新数据行，前一个事务再执行同一条 SELECT 语句时就可能多看到一个新的数据行，那就是一个幻影数据行。

为了解决这些问题，InnoDB 存储引擎提供了 4 种隔离级别。这些隔离级别用来确定允许某个事务看到与之同时执行的其他事务所作出的哪些修改，如下所示。

- READ UNCOMMITTED。允许某个事务看到其他事务尚未提交的数据行改动。
- READ COMMITTED。只允许某个事务看到其他事务已经提交的数据行改动。
- REPEATABLE READ。如果某个事务两次执行同一个 SELECT 语句，其结果是可重复的。换句话说，即使有其他事务在同时插入或修改数据行，这个事务所看到的结果也是一样的。
- SERIALIZABLE。这个隔离级别与 REPEATABLE READ 很相似，但对事务的隔离更彻底：某个事务正在查看的数据行不允许其他事务修改，直到该事务完成为止。换句话说，如果某个事务正在读取某些数据行，在它完成之前，其他事务将无法对那些数据行修改。

表 2-4 列出了这 4 种隔离级别以及它们是否允许脏读、不可重复读取或幻影数据行等问题。这个表格只适用于 InnoDB 存储引擎——REPEATABLE READ 隔离级别不能容忍幻影数据行。有些数据库系统的 REPEATABLE READ 隔离级别允许出现幻影数据行。

表2-4　隔离级别允许的问题

隔离级别	脏　读	不可重复读取	幻影数据行
READ UNCOMMITTED	是	是	是
READ COMMITTED	否	是	是
REPEATABLE READ	否	否	否
SERIALIZABLE	否	否	否

InnoDB 存储引擎默认使用的隔离级别是 REPEATABLE READ。这可以通过在启动服务器时使用 --transaction-isolation 选项或在服务器运行时使用 SET TRANSACTION 语句来改变。该语句有 3 种形式：

```
SET GLOBAL TRANSACTION ISOLATION LEVEL level;
SET SESSION TRANSACTION ISOLATION LEVEL level;
SET TRANSACTION ISOLATION LEVEL level;
```

SUPER 权限的客户可以使用 SET TRANSACTION 语句改变全局隔离级别的设置，该设置将作用于此后连接到服务器的任何客户。此外，任何客户都可以修改它自己的事务隔离级别，用 SET SESSION TRANSACTION 语句做出的修改将作用于在与服务器的本次会话里后续的所有事务，用 SET TRANSACTION 语句做出的修改只作用于下一个事务。客户在修改它自己的隔离级别时不需要任何特殊的权限。

本节里的绝大多数信息也适用于 Falcon 存储引擎。Falcon 和 InnoDB 存储引擎在这方面的主要区别是：Falcon 不支持 READ UNCOMMITTED 隔离级别，它目前也不支持 SERIALIZABLE 隔离级别（但 Falcon 开发团队正在为此而努力着）。

2.13.4　事务问题的非事务解决方案

在一个不支持事务的环境里，有些事务问题可以想办法解决，有些问题则毫无办法。下面将讨论哪些问题可以、哪些问题不可以在不使用事务的情况下得到解决。你可以利用这些信息去判断是否可

以把这里介绍的技巧用在某个应用程序里，以避免因为使用具备事务安全性的数据表而增加开销。

首先看看当有多个客户试图使用一些需要多条语句才能完成的操作去修改同一个数据库时，都会导致哪些并发问题。假设你开了一个服装店，你的销售管理软件会在售货员录入一份成交单据时自动更新相应的库存记录。下面是在同一时间卖出多件服装时可能发生的问题。为了便于描述，不妨假设你的衬衫库存记录的初始值是 47 件。

(1) 售货员 A 卖出了 3 件衬衫并把单据录入了电脑。销售管理软件开始更新数据库，它首先要选取当前的衬衫库存数量：

```
SELECT quantity FROM inventory WHERE item = 'shirt';
```

(2) 与此同时，售货员 B 也卖出了两件衬衫并把单据录入了电脑。第二台电脑里的软件也开始更新数据库：

```
SELECT quantity FROM inventory WHERE item ='shirt';
```

(3) 第一台电脑计算出新的库存数量是 47−3=44 件，并对仓库里的衬衫数量作出了相应的修改：

```
UPDATE inventory SET quantity = 44 WHERE item = 'shirt';
```

(4) 第二台电脑计算出新的库存数量是 47−2=45 件，并对仓库里的衬衫数量作出了相应的修改：

```
UPDATE inventory SET quantity = 45 WHERE item = 'shirt';
```

在这一系列事情结束之后，你的售货员总共卖出了 5 件衬衫，这是个好消息。可是，衬衫的库存数量是 45 件，这就不对了，它应该是 42 件。导致这一问题的根源在于这里有一个多语句操作：一条语句用来查询库存数量，另一条语句用来更新这个值。发生在第二条语句里的动作依赖于第一条语句检索出来的值。如果不同的多语句操作在发生时间上出现重叠，它们就有可能彼此交叉和干扰。要想解决这个问题，就必须设法保证每个多语句操作在执行时都不会与其他的操作发生干扰。

明确地锁定数据表。把多条语句用 LOCK TABLES 和 UNLOCK TABLES 语句括起来就可以把它们当做一个单元来执行：锁定需要使用的所有数据表，发出你的语句，解除锁定。这可以防止其他人在你锁定有关数据表期间修改它们。利用这种锁定机制完成前面的衬衫库存数量更新操作的过程如下所述。

(1) 售货员 A 卖出了 3 件衬衫并把单据录入了电脑。你的销售管理软件开始更新数据库，它先锁定数据表并选取当前的衬衫库存数量（47）：

```
LOCK TABLES inventory WRITE;
SELECT quantity FROM inventory WHERE item = 'shirt';
```

这里需要使用 WRITE 锁，因为整个操作的最终目的是修改 inventory 数据表，需要对它进行写操作。

(2) 与此同时，售货员 B 也卖出了两件衬衫并把单据录入了电脑。第二台电脑里的软件也开始更新数据库，它也是先从锁定数据表开始：

```
LOCK TABLES inventory WRITE;
```

具体到这个例子，这条语句将被阻塞，因为售货员 A 已经锁定那个数据表了。

(3) 第一台电脑计算出新的库存数量是 47−3=44 件，它更新了衬衫件数并解除了锁定：

```
UPDATE inventory SET quantity = 44 WHERE item = 'shirt';
UNLOCK TABLES;
```

(4) 在第一台电脑解除了对数据表的锁定之后，第二台电脑的锁定请求成功了，它继续执行并检索出当前衬衫库存数量是 44 件：

```
SELECT quantity FROM inventory WHERE item = 'shirt';
```

(5) 第二台电脑计算出新的库存数量是 47−2=45 件，它更新了衬衫件数并解除了锁定：

```
UPDATE inventory SET quantity = 42 WHERE item = 'shirt';
UNLOCK TABLES;
```

就这样，来自两个操作的各语句不再相互混杂，对库存数量的修改也就不会再出现错误了。

如果你正在使用多个数据表，在执行多语句操作之前必须把它们全部锁定。如果你只从某个特定的数据表读取数据，你只需要给它加上一个“读操作”锁就行了，用不着给它加上“写操作”锁。（“读操作”锁允许其他客户在你使用被锁定数据表时读取它，但不允许对之进行写操作。）比如说，假设你有一组查询是用来修改 inventory 数据表的，你还需要从 customer 数据表读取一些数据。此时，你需要给 inventory 数据表加上一把“写操作”锁，给 customer 数据表加上一把“读操作”锁：

```
LOCK TABLES inventory WRITE, customer READ;
... use the tables here ...
UNLOCK TABLES;
```

使用相对更新操作，不使用绝对更新操作。在先明确锁定数据表再更新库存的办法里，整个操作需要两条语句来完成，一条语句用来查询当前库存水平，另一条语句用来计算本次销售后的衬衫库存数量并对数据表进行相应的更新。让多个客户同时进行的操作互不干扰的另一个办法，是把整个操作压缩成只用一条语句来完成。这将消除多语句操作中各条语句之间的彼此依赖。并非所有的操作都可以只用一条语句完成，但就刚才那个更新衬衫库存数量的例子而言，这个策略是可行的。只要把计算衬衫库存数量的语句修改为使用相对于它的当前值就可以只用一个步骤完成库存更新操作了，如下所示。

(1) 售货员 A 卖出了 3 件衬衫，销售管理软件把衬衫的当前库存数量减去 3：

```
UPDATE inventory SET quantity = quantity - 3 WHERE item = 'shirt';
```

(2) 售货员 B 卖出了两件衬衫，销售管理软件把衬衫的当前库存数量减去 2：

```
UPDATE inventory SET quantity = quantity - 2 WHERE item = 'shirt';
```

这个办法的优点是数据库修改时不再需要使用多条语句。这消除了并发问题，所以不再需要明确地锁定数据表。如果你打算执行的某个操作与此相似，也许根本用不着借助事务机制就可以完成它。

刚才介绍的这两种非事务解决方案可以解决许多实际问题，但它们也有一定的局限性，如下所示。

- 并非所有的操作都可以被编写成一条相对更新语句。有些问题只有使用多条语句才能解决，你必须考虑和解决随之而来的并发问题。
- 在多语句操作期间锁定数据表可以避免客户之间相互干扰，但万一在操作过程中出现错误又会发生什么样的事情？此时，你肯定希望能够撤销此前已执行完毕的语句对数据库的影响，不让改了一半儿的数据留在数据库里造成数据库的不稳定。令人遗憾的是，虽然数据表锁定机制可以帮助你解决并发问题，但如果所涉及的数据表不支持事务处理，它们也不能为你的善后恢复工作提供多少帮助。
- 锁定机制需要你锁定和释放数据表。如果你改用了其他的数据表，千万不要忘记对 LOCK TABLES 语句作相应的修改。

如果这些问题对你的应用程序很重要，你应该选用具备事务安全性的数据表，事务机制可以帮你妥善处理所有这些问题。事务机制可以把一组语句当做一个不可分割的单元来执行，并防止客户之间彼此干扰，从而有效地管理好并发问题。它提供的回滚功能还可以在发生意外时避免尚未全部完成的操作损坏数据库。它还可以自行判断并获得必要的操作锁，让你不再为这些细节操心费力。

事务数据表和非事务数据表可以混用吗

在某次事务中混合使用事务数据表和非事务数据表是允许的，但最终的结果不一定符合你的期望。对非事务数据表进行操作的语句总是立刻生效，即便是自动提交模式处于禁用状态也是如此。事实上，非事务数据表永远待在自动提交模式下，每条语句都会在执行完毕后立刻提交。因此，如果你在一个事务里修改了一个非事务数据表，那么这个修改将无法撤销。

2.14 外键和引用完整性

利用外键（foreign key）关系可以在某个数据表里声明与另一个数据表里的某个索引相关联的索引。还可以把你想施加在数据表上的约束条件放到外键关系里，让系统根据这个关系里的规则来维护数据的引用完整性。比如说，sampdb 数据库里的 score 数据表包含一个 student_id 数据列，我们要用它把 score 数据表里的考试成绩与 student 数据表里的学生联系在一起。当我们在第 1 章里创建这些数据表时，我们在它们之间建立了一些明确的关系，其中之一是把 score.student_id 数据列定义为 student.student_id 数据列的一个外键。这可以确保只有那些在 student 数据表里存在 student_id 值的数据行才能被插入到 score 数据表里。换句话说，这个外键可以确保不会出现为一名并不存在的学生输入了成绩的错误。

外键不仅在数据行的插入操作中很有用，在删除和更新操作中也很有用。比如说，我们可以建立这样一个约束条件：在把某个学生从 student 数据表里删除时，score 数据表里与这个学生有关的所有数据行也将自动被删除。这被称为级联删除（cascaded delete），因为删除操作的效果就像瀑布（cascade）那样从一个数据表“流淌”到另外一个数据表。级联更新也是可能的。比如说，如果利用瀑布式更新在 student 数据表里改变了某个学生的 student_id，score 数据表里与这个学生相对应的所有数据行的这个值也应该发生相应的改变。

外键可以帮我们维护数据的一致性，它们用起来也很方便。如果不使用外键，就必须由你来负责保证数据表之间的依赖关系和维护它们的一致性，而这意味着你的应用程序必须增加一些必要的代码。在某些情况下，这只需要你额外发出几条 DELETE 语句以确保当你删除某个数据表里的数据行时，其他数据表里与之相对应的数据行也将随之一起被删除。但额外工作毕竟是额外工作，而且既然数据库引擎能够替你进行数据一致性检查，为什么不让它干呢？要是你的数据表有非常复杂的关系，由你在你的应用程序里通过代码去检查这些依赖关系就会变得很麻烦，而数据库系统提供的自动检查能力往往要比你本人考虑得更周全和更细致，也更简明实用。

在 MySQL 里，InnoDB 存储引擎提供了外键支持。本节将讨论如何设置 InnoDB 数据表以定义外键以及外键将如何影响你使用数据表的方式和方法。首先，有必要定义几个术语：

- 父表，包含原始键值的数据表；
- 子表，引用父表中的键值的相关数据表。

父表中的键值用来关联两个数据表。具体地说，子表中的某个索引引用父表中的某个索引。子表

的索引值必须匹配父表中的索引值或是被设置为 NULL 以表明在父表里不存在与之对应的数据行。子表里的索引就是所谓的“外键”，因为它们存在于父表的外部，但包含指向父表的值。外键关系可以被设置成不允许使用 NULL 值，此时所有的外键值都必须匹配父表里的某个值。

InnoDB 存储引擎通过这些规则来保证在外键关系里不会有不匹配的东西，这被称为引用完整性（referential integrity）。

2.14.1　外键的创建和使用

在子表里定义一个外键的语法如下所示：

```
[CONSTRAINT constraint_name]
FOREIGN KEY [fk_name] (index_columns)
  REFERENCES tbl_name (index_columns)
  [ON DELETE action]
  [ON UPDATE action]
  [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]
```

这个语法本身很完备，但 InnoDB 存储引擎目前还没有实现所有的子句：它目前还不支持 MATCH 子句，即使你给出了一条 MATCH 子句，它也会被忽略。有几种 action 值目前只能被识别出来，但不会有任何效果。（除 InnoDB 以外的其他存储引擎在遇到 FOREIGN KEY 定义时不会报告错误，但会把它整个忽略掉。）

InnoDB 存储引擎目前能够识别和支持以下外键定义语法成分。

- CONSTRAINT 子句。如果给出，这个子句用来给外键约束关系起一个名字。如果你省略了它，InnoDB 存储引擎将创建一个名字。
- FOREIGN KEY 子句。列出子表里的被索引数据列，它们必须匹配父表里的索引值。fk_name 是外键的 ID。如果你给出了一个 fk_name，在 InnoDB 存储引擎能够为外键自动创建一个索引的情况下它将成为那个索引的名字；否则，它将被忽略。
- REFERENCES 子句。列出父表和父表中的索引数据列的名字，子表里的外键将引用这个子句所列出的父表数据列。在 REFERENCES 子句的 index_columns 部分列出的数据列的个数必须与在 FOREIGN KEY 子句的 index_columns 部分列出的数据列的个数相同。
- ON DELETE 子句。用来设定在父表里的数据行被删除时子表应该发生什么事。如果没有 ON DELETE 子句，其默认行为是拒绝从父表里删除仍有子表数据行在引用它们的数据行。如果你想明确地指定一种 action 值，请使用以下子句之一。
  - ON DELETE NO ACTION 和 ON DELETE RESTRICT 子句。它们的含义与省略 ON DELETE 子句一样。（有些数据库系统提供了延迟检查功能，而 NO ACTION 是一种延迟检查。在 MySQL 里，外键约束条件是被立刻检查的，所以 NO ACTION 和 RESTRICT 的含义完全一样。）
  - ON DELETE CASCADE 子句。在删除父表数据行时，子表里与之相关联的数据行也将被删除。本质上，删除效果将从父表蔓延到子表。这样一来，你只需删除父表里的数据行并让 InnoDB 存储引擎负责从子表删除相关数据行，就可以完成一个涉及多个数据表的删除操作了。
  - ON DELETE SET NULL 子句。在删除父表数据行时，子表里与之相关联的索引列将被设置为 NULL。如果你打算使用这个选项，就必须把在外键定义里列出的所有被编制索引的子表数据列定义为允许 NULL 值。（使用这个动作的一个隐含限制是你不能把外键定义为 PRIMARY KEY，因为主键不允许 NULL 值。）

  - ■ ON DELETE SET DEFAULT 子句。这个子句可以被识别出来，但目前尚未实现；InnoDB 存储引擎在遇到这个子句时将报告一个错误。
- ❑ ON UPDATE 子句。用来设定当父表里的数据行更新时子表应该发生什么事。如果没有 ON UPDATE 子句，其默认行为是拒绝插入或更新其外键值在父表索引里没有任何匹配的子表数据行，并阻止仍有子表数据行在引用着它们的父表索引值被更新。可供选用的 action 值及其效果与 ON DELETE 子句的相同。

如果你想建立一个外键关系，请遵守以下指示。

- ❑ 子表必须有这样一个索引。在定义该索引时，必须首先列出外键数据列。父表必须有这样一个索引：在定义该索引时，必须首先列出 REFERENCES 子句里的数据列。（换句话说，外键里的数据列在外键关系所涉及的两个数据表里都必须有索引。）在定义外键关系之前，你必须明确地创建出必要的父表索引。InnoDB 存储引擎将自动地在子表里为外键数据列（引用数据列）创建一个索引——如果在你用来创建子表的 CREATE TABLE 语句里没有包括一个这样的索引的话。不过，由 InnoDB 存储引擎自动创建的这种索引将是一个非唯一的索引，并且只包含外键数据列。如果你想让这个子表索引是一个 PRIMARY KEY 或 UNIQUE 索引，或者如果你想让它在外键数据列之外还包括其他的数据列，就只能由你本人明确地定义它了。
- ❑ 父表和子表索引里的对应数据列必须是兼容的数据类型。比如说，你不能让一个 INT 数据列去匹配一个 CHAR 数据列。对应的字符数据列必须是同样的长度。对应的整数数据列必须是同样的尺寸，并且必须要么都带符号，要么都被定义成 UNSIGNED。
- ❑ 你不能对外键关系里的字符串数据列的前缀编制索引。换句话说，对于字符串数据列，你必须对整个数据列编制索引，不能只对它的前几个字符编制索引。

在第 1 章里，我们为考试成绩管理项目创建了几个有着简单外键关系的数据表。我们现在来看一个比较复杂的例子。首先创建两个分别名为 parent 和 child 的数据表，其中 child 数据表包含一个外键，该外键引用 parent 数据表里的 par_id 数据列：

```
CREATE TABLE parent
(
  par_id    INT NOT NULL,
  PRIMARY KEY (par_id)
) ENGINE = INNODB;

CREATE TABLE child
(
  par_id    INT NOT NULL,
  child_id  INT NOT NULL,
  PRIMARY KEY (par_id, child_id),
  FOREIGN KEY (par_id) REFERENCES parent (par_id)
    ON DELETE CASCADE
    ON UPDATE CASCADE
) ENGINE = INNODB;
```

这个例子在定义外键时使用了 ON DELETE CASCADE 子句，它指定当 parent 数据表里的某个数据行被删除时，MySQL 将自动地从 child 数据表里把有匹配 par_id 值的数据行也删掉。ON UPDATE CASCADE 子句表明：如果 parent 数据表里的某个数据行的 par_id 值被改变了，MySQL 将自动地把 child 数据表里的所有匹配的 par_id 值也改成新值。

现在，在 parent 数据表里插入一些数据行，再在 child 数据表插入一些有着相关键值的数据行：

```
mysql> INSERT INTO parent (par_id) VALUES(1),(2),(3);
mysql> INSERT INTO child (par_id,child_id) VALUES(1,1),(1,2);
mysql> INSERT INTO child (par_id,child_id) VALUES(2,1),(2,2),(2,3);
mysql> INSERT INTO child (par_id,child_id) VALUES(3,1);
```

这些语句将导致如下所示的数据表内容，而 child 数据表里的每一个 par_id 值都分别匹配 parent 数据表里的一个 par_id 值：

```
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      1 |
|      2 |
|      3 |
+--------+
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|      1 |        1 |
|      1 |        2 |
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

为了证明 InnoDB 存储引擎在插入新数据行时会遵守外键关系的约束，我们现在故意往 child 数据表插入一个“错误的”数据行，它的 par_id 值在 parent 数据表里没有匹配：

```
mysql> INSERT INTO child (par_id,child_id) VALUES(4,1);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails (`sampdb`.`child`, CONSTRAINT `child_ibfk_1` FOREIGN
KEY (`par_id`) REFERENCES `parent` (`par_id`) ON DELETE CASCADE
ON UPDATE CASCADE)
```

想看看级联删除的效果？从 parent 数据表删除一个数据行试试：

```
mysql> DELETE FROM parent WHERE par_id = 1;
```

MySQL 将从父表删除这个数据行：

```
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      2 |
|      3 |
+--------+
```

同时，MySQL 还将把这个 DELETE 语句的效果蔓延到 child 数据表：

```
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
```

```
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

想看看级联更新的效果？对 parent 数据表里的某个数据行更新一下试试：

```
mysql> UPDATE parent SET par_id = 100 WHERE par_id =2;
mysql> SELECT * FROM parent;
+--------+
| par_id |
+--------+
|      3 |
|    100 |
+--------+
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|      3 |        1 |
|    100 |        1 |
|    100 |        2 |
|    100 |        3 |
+--------+----------+
```

上面几个例子演示了 parent 数据表里的数据行删除和更新操作将导致 child 数据表里的相关数据行被级联删除或更新的情况。ON DELETE 和 ON UPDATE 子句还支持其他动作。比如说，你可以把 child 数据表里的数据行保留下来不删除，但它们的外键数据列将被设置为 NULL。要想做到这一点，你必须对 child 数据表的定义作一些必要的修改，如下所示。

- 使用 ON DELETE SET NULL 来代替 ON DELETE CASCADE。这将使 InnoDB 存储引擎把外键数据列（par_id）设置为 NULL 而不是删除那些数据列。
- 使用 ON UPDATE SET NULL 来代替 ON UPDATE CASCADE。这将使 InnoDB 存储引擎在 parent 数据表里的数据行被更新时把 child 数据表里的对应数据行的外键数据列（par_id）设置为 NULL。
- child 数据表里的 par_id 数据列最初被定义成 NOT NULL。这不能与 ON DELETE SET NULL 或 ON UPDATE SET NULL 配合使用，所以必须把这个数据列的定义改成允许有 NULL 值。
- child 数据表里的 par_id 数据列最初也被定义成 PRIMARY KEY 的一部分。因为 PRIMARY KEY 不允许包含 NULL 值，所以在把 child 数据表里的 par_id 数据列改成允许为 NULL 值的同时，还需要把那个 PRIMARY KEY 改成一个 UNIQUE 索引。UNIQUE 索引要求索引值必须是独一无二的——但 NULL 值除外，NULL 值可以在索引里出现多次。

想看看这些修改的效果吗？按最初的定义重新创建 parent 数据表并把同样的数据行加载到其中，然后用如下所示的新定义创建一个新的 child 数据表：

```
CREATE TABLE child
(
  par_id    INT NULL,
  child_id  INT NOT NULL,
  UNIQUE (par_id, child_id),
```

```
  FOREIGN KEY (par_id) REFERENCES parent (par_id)
    ON DELETE SET NULL
    ON UPDATE SET NULL
) ENGINE = INNODB;
```

现在，在往child数据表插入新数据行时，child数据表的行为和原来的定义基本一样：只有其par_id值在parent数据表里出现过的数据行才能被插入child数据表，否则将拒绝插入：

```
mysql> INSERT INTO child (par_id,child_id) VALUES(1,1),(1,2);
mysql> INSERT INTO child (par_id,child_id) VALUES(2,1),(2,2),(2,3);
mysql> INSERT INTO child (par_id,child_id) VALUES(3,1);
mysql> INSERT INTO child (par_id,child_id) VALUES(4,1);
ERROR 1452 (23000): Cannot add or update a child row: a foreign key
constraint fails ('sampdb'.'child', CONSTRAINT 'child_ibfk_1' FOREIGN
KEY ('par_id') REFERENCES 'parent' ('par_id') ON DELETE SET NULL
ON UPDATE SET NULL)
```

请注意，往child数据表插入新数据行时与原来有一点区别：因为par_id数据列现在被定义成允许为NULL值，所以你现在可以把包含NULL值的新数据行插入child数据表而不会引起错误。另一个区别体现在从parent数据表删除数据行的时候，从parent数据表删除一个数据行，然后查看一下child数据表的内容就知道怎么回事了：

```
mysql> DELETE FROM parent WHERE par_id = 1;
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|   NULL |        1 |
|   NULL |        2 |
|      2 |        1 |
|      2 |        2 |
|      2 |        3 |
|      3 |        1 |
+--------+----------+
```

看到了吗？child数据表里par_id数据列的值是1的数据行没有被删掉，但它们的par_id数据列的值都被设置成了NULL，这正是ON DELETE SET NULL子句的效果。

对parent数据表里的数据行进行刷新将有类似的效果：

```
mysql> UPDATE parent SET par_id = 100 WHERE par_id = 2;
mysql> SELECT * FROM child;
+--------+----------+
| par_id | child_id |
+--------+----------+
|   NULL |        1 |
|   NULL |        1 |
|   NULL |        2 |
|   NULL |        2 |
|   NULL |        3 |
|      3 |        1 |
+--------+----------+
```

如果你想查看某个 InnoDB 数据表都有哪些外键关系，可以使用 SHOW CREATE TABLE 或 SHOW TABLE STATUS 语句。

如果你在试图创建一个带有外键关系的数据表时遇到问题，可以使用 SHOW ENGINE INNODB STATUS 语句查看完整的出错消息。

2.14.2 如果不能使用外键该怎么办

如果 MySQL 服务器没有 InnoDB 支持，或者如果你正在使用另一种存储引擎（因为 InnoDB 不能提供你需要的功能，如 FULLTEXT 索引或空间数据类型），你就享受不到外键带来的好处。在这类情况下，怎样才能保证数据表之间的关系是正确和稳定的呢？

外键施加的约束条件并不难通过编程逻辑来实现。有时候，只要换个办法来录入数据就可以解决问题。以考试成绩管理项目中的 student 和 score 数据表为例，它们现在是通过一个基于每个表中 student_id 值的外键关系关联在一起。如果我们当初把它们创建为 MyISAM 数据表而不是 InnoDB 数据表，因为 MyISAM 数据表不支持外键，这两个数据表之间的关系将是隐式的而不是显式的。在某次考试或小测验之后，你将需要把一组新的考试成绩添加到数据库里，而你必须保证不会把其 student_id 值没列在 student 数据表里的 score 数据行添加进来。

从某种意义上讲，这只不过是采用正确办法录入数据。在需要输入一组新的考试成绩时，为了避免给一名并不存在的学生输入考试成绩，应该考虑编写一个应用小程序来帮助你：这个小程序依次列出 student 数据表里的每位学生，读取每名学生的考试成绩，使用该名学生的 ID 为 score 数据表生成一个新数据行。这个办法可以确保你永远也不会为一名并不存在的学生输入一个新数据行。不过，如果是由你本人手动发出一系列 INSERT 语句的话，就仍有可能出现“坏”数据行，而 InnoDB 数据表和外键可以保证不会发生这种错误。

从 student 数据表删除一个数据行该怎么办？假设你想删除编号是 13 的学生，这意味着应该从 score 数据表里把与这名学生有关的所有数据行都删掉。如果这两个数据表之间有一个指定级联删除的外键关系，你只要用如下所示的语句删除 student 数据表里的数据行就没事了，MySQL 将替你完成从 scorer 数据表删除相关数据行的工作：

```
DELETE FROM student WHERE student_id = 13;
```

在没有外键支持的情况下，要想获得与 ON DELETE CASCADE 同样的效果，就必须由你本人明确地在所有相关的数据表里把所有相关的数据行都删除干净才行：

```
DELETE FROM student WHERE student_id = 13;
DELETE FROM score WHERE student_id = 13;
```

另一个办法是使用一个涉及多个数据表的删除语句，这可以让你只用一条语句就获得与 ON DELETE CASCADE 同样的效果。但这里需要注意一个不容易察觉的陷阱。比如说，下面这条语句乍看起来毫无问题，其实却考虑得不够周全：

```
DELETE student, score FROM student INNER JOIN score
ON student.student_id = score.student_id WHERE student.student_id = 13;
```

这条语句的问题是：万一被删除的学生在 score 数据表里没有任何成绩记录，它将执行失败。此时，因为 WHERE 子句找不到任何匹配，所以 student 数据表不会有任何数据行被删除。具体到这个例子，LEFT JOIN 更适用，因为它可以把 student 数据表里应该被删除的的数据行全都找出来，包括那些在 score 数据表里没有匹配数据行：

```
DELETE student, score FROM student LEFT JOIN score USING (student_id)
WHERE student.student_id = 13;
```

2.15　使用 FULLTEXT 索引

MySQL 具备全文搜索的能力。全文搜索引擎可以在不使用模板匹配操作的情况下查找单词或短语。全文搜索分为 3 种模式，如下所示。

- 自然语言模式。把搜索字符串解释为一系列单词并查找包含这些单词的数据行。
- 布尔模式。把搜索字符串解释为一系列单词，但允许使用一些操作符字符来“修饰”这些单词以表明特定的要求，如某给定单词必须出现（或不出现）在匹配数据行里，某个数据行必须包含一个精确的短语，等等。
- 查询扩展模式。这种搜索分两阶段进行。第一阶段是自然语言搜索，第二阶段使用原来的搜索字符串加上在第一次搜索中找到的相关度最高的匹配数据行再进行一次搜索。这扩大了搜索范围，它可以把与原来的搜索字符串相关、但用原来的搜索字符串匹配不到的数据行也找出来。

要想对某个数据表进行全文搜索，必须事先为它创建一个 FULLTEXT 索引，这种索引具有以下特点。

- 全文搜索的基础是 FULLTEXT 索引，这种索引只能在 MyISAM 数据表里创建。FULLTEXT 索引只能由 CHAR、VARCHAR 和 TEXT 这几种类型的数据列构成。
- 全文搜索将忽略“常见的”单词，而“常见”在这里的含义是“至少在一半的数据行里出现过”。千万不要忘记这个特点，尤其是在你准备对数据表进行全文搜索测试时。你至少要在测试数据表里插入 3 个数据行。如果那个数据表只有一个或两个数据行，它里面的每个单词将至少有 50%的出现几率，所以对它进行全文搜索将不会有任何结果。
- 全文搜索还将忽略一些常用单词，如“the”、“after”和“other”等，这些单词被称为“休止单词”，MySQL 在进行全文搜索时总是会忽略它们。
- 太短的单词也将被忽略。在默认的情况下，“太短”指少于 4 个字符。但你可以通过重新配置服务器的办法把这个最小长度设置为其他值。
- 全文搜索对“单词”的定义是：由字母、数字、撇号（如“it’s”中的“’”）和下划线字符构成的字符序列。这意味着字符串“full-blood”将被解释为包含“full”和“blood”两个单词。全文搜索匹配整个单词，而不是单词的一部分。只要在一个数据行里找到了搜索字符串里的任何单词，FULLTEXT 引擎就会认为这个数据行与搜索字符串是匹配的。在此基础上，布尔式全文搜索还允许你加上一些额外的要求，比如说，所有的单词都必须出现（不论顺序）才认为是匹配，（在搜索一条短语时）单词顺序必须与在搜索字符串里列出的一致，等等。布尔搜索还可以用来匹配不包含特定单词的数据行，或者通过添加一个通配符来匹配以一个给定前缀开头的所有单词。
- FULLTEXT 索引可以为一个或多个数据列而创建。如果它涉及多个数据列，基于该索引的搜索将在所有数据列上同时进行。反过来说，在进行全文搜索时，你给出的数据列清单必须和某个 FULLTEXT 索引所匹配的那些数据列精确匹配。比如说，如果你需要分别搜索 col1、col2 以及“col1 和 col2”，你将需要创建 3 个索引：col1 和 col2 各有一个，“col1 和 col2”有一个。

接下来的例子演示了全文搜索的使用方法。我们将先创建几个 FULLTEXT 索引，然后用 MATCH 操作符对它们进行一些查询。用来创建数据表并把一些样板数据加载到其中的脚本可以在 sampdb 数据

库的 fulltext 子目录里找到。

FULLTEXT 索引的具体创建过程与其他索引大同小异。你可以在创建一个新数据表的同时在 CREATE TABLE 语句里定义它们，也可以在数据表被创建出来以后再用 ALTER TABLE 或 CREATE INDEX 语句添加它们。因为 FULLTEXT 索引要求你必须使用 MyISAM 数据表，所以如果你正在创建一个需要使用全文搜索的 MyISAM 数据表，不妨利用一下 MyISAM 存储引擎的这个特点来加快点儿速度。在加载数据时，先填充数据表、再添加索引的办法要比先创建索引再加载数据的办法快得多。假设你有一个名为 apothegm.txt 的数据文件，其内容是一些名人名言：

```
Aeschylus               Time as he grows old teaches many lessons
Alexander Graham Bell   Mr. Watson, come here. I want you!
Benjamin Franklin       It is hard for an empty bag to stand upright
Benjamin Franklin       Little strokes fell great oaks
Benjamin Franklin       Remember that time is money
Miguel de Cervantes     Bell, book, and candle
Proverbs 15:1           A soft answer turneth away wrath
Theodore Roosevelt      Speak softly and carry a big stick
William Shakespeare     But, soft! what light through yonder window breaks?
Robert Burton           I light my candle from their torches.
```

如果按“名人”、“名言”和“名人加名言”进行搜索，你需要创建 3 个 FULLTEXT 索引：两个数据列各有一个，它们加起来有一个。下面这些语句将创建、填充一个名为 apothegm 的数据表，并为它编制索引：

```
CREATE TABLE apothegm (attribution VARCHAR(40), phrase TEXT) ENGINE = MyISAM;
LOAD DATA LOCAL INFILE 'apothegm.txt' INTO TABLE apothegm;
ALTER TABLE apothegm
  ADD FULLTEXT (phrase),
  ADD FULLTEXT (attribution),
  ADD FULLTEXT (phrase, attribution);
```

2.15.1 全文搜索：自然语言模式

把数据表创建出来之后，对它进行自然语言模式的全文搜索：用 MATCH 操作符列出将被搜索的数据列、用 AGAINST()给出搜索字符串。如下所示：

```
mysql> SELECT * FROM apothegm WHERE MATCH(attribution) AGAINST('roosevelt');
+--------------------+------------------------------------+
| attribution        | phrase                             |
+--------------------+------------------------------------+
| Theodore Roosevelt | Speak softly and carry a big stick |
+--------------------+------------------------------------+
mysql> SELECT * FROM apothegm WHERE MATCH(phrase) AGAINST('time');
+-------------------+-------------------------------------------+
| attribution       | phrase                                    |
+-------------------+-------------------------------------------+
| Benjamin Franklin | Remember that time is money               |
| Aeschylus         | Time as he grows old teaches many lessons |
+-------------------+-------------------------------------------+
mysql> SELECT * FROM apothegm WHERE MATCH(attribution, phrase)
    -> AGAINST('bell');
```

```
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
| Miguel de Cervantes   | Bell, book, and candle            |
+-----------------------+-----------------------------------+
```

在最后一个例子里，请注意查询是如何在不同的数据列里找出包含搜索单词的数据行的，它展示了利用 FULLTEXT 索引同时搜索多个数据列的能力。还请注意，我们在查询命令里列出数据列的顺序是 attribution, phrase，而在创建索引时用的是（phrase, attribution）；这是为了告诉你这个顺序不重要。重要的是必须有一个 FULLTEXT 索引精确地包含你在查询命令里列出的数据列。

如果只想看看某个搜索可以匹配到多少数据行，请使用 COUNT(*)：

```
mysql> SELECT COUNT(*) FROM apothegm WHERE MATCH(phrase) AGAINST('time');
+----------+
| COUNT(*) |
+----------+
|        2 |
+----------+
```

当你在 WHERE 子句里使用 MATCH 表达式时，自然语言模式的全文搜索的输出数据行按照相关程度递减的顺序排序。相关度以非负浮点数来表示，零代表“毫不相关”。要想查看这些值，在输出数据列清单里加上一个 MATCH 表达式即可：

```
mysql> SELECT phrase, MATCH(phrase) AGAINST('time') AS relevance
    -> FROM apothegm;
+-----------------------------------------------------+-----------------+
| phrase                                              | relevance       |
+-----------------------------------------------------+-----------------+
| Time as he grows old teaches many lessons           | 1.3253291845322 |
| Mr. Watson, come here. I want you!                  |               0 |
| It is hard for an empty bag to stand upright        |               0 |
| Little strokes fell great oaks                      |               0 |
| Remember that time is money                         | 1.3400621414185 |
| Bell, book, and candle                              |               0 |
| A soft answer turneth away wrath                    |               0 |
| Speak softly and carry a big stick                  |               0 |
| But, soft! what light through yonder window breaks? |               0 |
| I light my candle from their torches.               |               0 |
+-----------------------------------------------------+-----------------+
```

自然语言模式的搜索能够找到包含任何一个搜索单词的数据行，所以下面这个查询将把包含单词“hard”或“soft”的数据行都找出来：

```
mysql> SELECT * FROM apothegm WHERE MATCH(phrase)
    -> AGAINST('hard soft');
+---------------------+-----------------------------------------------------+
| attribution         | phrase                                              |
+---------------------+-----------------------------------------------------+
| Benjamin Franklin   | It is hard for an empty bag to stand upright        |
| Proverbs 15:1       | A soft answer turneth away wrath                    |
| William Shakespeare | But, soft! what light through yonder window breaks? |
+---------------------+-----------------------------------------------------+
```

自然语言模式是默认的全文搜索模式，在 MySLQ 5.1 和更高版本里，可以通过在搜索字符串的后

面加上 IN NATURAL LANGUAGE MODE 的办法明确地指定这个模式。下面这条语句和前一个例子做的事情完全一样。

```
SELECT * FROM apothegm WHERE MATCH(phrase)
AGAINST('hard soft' IN NATURAL LANGUAGE MODE);
```


2.15.2 全文搜索：布尔模式

全文搜索的布尔模式可以让我们控制多单词搜索操作中的许多细节。要想进行这种模式的搜索，需要在 AGAINST()函数里在搜索字符串的后面加上 IN BOOLEAN MODE 短语。布尔模式的全文搜索有以下特点。

- "50%规则"不再起作用，即使是在超过一半的数据行里出现过的单词也可以被这种搜索匹配出来。
- 查询结果不再按照相关程度排序。
- 在搜索一个短语时，你可以要求所有单词必须按照某种特定的顺序出现。如果是搜索一个短语，需要把构成该短语的所有单词用双引号括起来；如果数据行包含的单词按照给定的顺序排列，才被认为是一个匹配。

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('"bell book and candle"' IN BOOLEAN MODE);
+---------------------+------------------------+
| attribution         | phrase                 |
+---------------------+------------------------+
| Miguel de Cervantes | Bell, book, and candle |
+---------------------+------------------------+
```

- 布尔模式的全文搜索还可以在没被包括在FULLTEXT索引里的数据列上进行，但这要比搜索有FULLTEXT索引的数据列慢很多。

在进行布尔搜索时，还可以给搜索字符串里的单词加上一些修饰符。在单词的前面加上一个加号表示该单词必须出现在匹配数据行里，而加上一个减号表示该单词不得出现在匹配数据行里。比如说，搜索字符串'bell'匹配包含"bell"的数据行，而在布尔模式里，搜索字符串'+bell -candle'只匹配包含单词"bell"、不包含单词"candle"的数据行：

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell');
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
| Miguel de Cervantes   | Bell, book, and candle            |
+-----------------------+-----------------------------------+
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('+bell -candle' IN BOOLEAN MODE);
+-----------------------+-----------------------------------+
| attribution           | phrase                            |
+-----------------------+-----------------------------------+
```

```
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
+-----------------------+------------------------------------+
```

后缀的星号“*”将被解释为一个通配符，带星号后缀的搜索单词将匹配以它开头的所有单词。比如说，'soft*'将匹配“soft”、“softly”、“softness”等：

```
mysql> SELECT * FROM apothegm WHERE MATCH(phrase)
    -> AGAINST('soft*' IN BOOLEAN MODE);
+---------------------+-----------------------------------------------+
| attribution         | phrase                                        |
+---------------------+-----------------------------------------------+
| Proverbs 15:1       | A soft answer turneth away wrath              |
| William Shakespeare | But, soft! what light through yonder window breaks?|
| Theodore Roosevelt  | Speak softly and carry a big stick            |
+---------------------+-----------------------------------------------+
```

注意，星号通配符不能用来匹配比最小索引单词长度更短的单词。

在附录 C 里，你可以在介绍 MATCH 操作符的部分查到全部的布尔模式修饰符。

和自然语言模式的全文搜索情况相似，布尔模式的全文搜索也将忽略所有的休止单词，就算是把它们标为“必须出现”也是如此。比如说，搜索字符串'+Alexander +the +great'将找出包含“Alexander”和“great”的数据行，“the”会因为它是一个休止单词而被忽略。

2.15.3　全文搜索：查询扩展模式

全文搜索的查询扩展模式将进行两个阶段的搜索。第一遍搜索和普通的自然语言搜索一样，在这次搜索里找到的相关程度最高的数据行里的单词将被用在第二阶段。这些数据行里的单词加上原来那些搜索单词将被用来进行第二遍搜索。因为搜索单词的集合变大了，所以在最终结果里往往会多出一些在第一阶段没被找到、但与第一阶段的检索结果有一定关系的数据行。

要想进行这种搜索，需要在搜索字符串的后面加上 WITH QUERY EXPANSION 短语。下面的例子提供了一个演示。第一条查询命令将进行一次自然语言搜索。第二条查询命令将进行一次查询扩展搜索，这次多找到了一个数据行，但该数据行不包含原始搜索字符串里的任何单词。该数据行会被匹配出来的原因是它包含单词“candle”，这个单词出现在了被自然语言搜索找到的某个数据行里。

```
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell book');
+-----------------------+------------------------------------+
| attribution           | phrase                             |
+-----------------------+------------------------------------+
| Miguel de Cervantes   | Bell, book, and candle             |
| Alexander Graham Bell | Mr. Watson, come here. I want you! |
+-----------------------+------------------------------------+
mysql> SELECT * FROM apothegm
    -> WHERE MATCH(attribution, phrase)
    -> AGAINST('bell book' WITH QUERY EXPANSION);
+-----------------------+-----------------------------------------+
| attribution           | phrase                                  |
+-----------------------+-----------------------------------------+
| Miguel de Cervantes   | Bell, book, and candle                  |
| Alexander Graham Bell | Mr. Watson, come here. I want you!      |
```

```
| Robert Burton          | I light my candle from their torches. |
+------------------------+---------------------------------------+
```

2.15.4 配置全文搜索引擎

有几个全文搜索参数是可配置的，可以通过设置系统变量的办法来改变。用来为 FULLTEXT 索引设定单词最小和最大长度的参数是 ft_min_word_len 和 ft_max_word_len。长度超出这两个参数所定义的范围的单词在创建 FULLTEXT 索引时将被忽略。默认的最小值和最大值分别是 4 个和 84 个字符。

假设你想把最小单词长度从 4 改成 3，请按以下步骤进行。

(1) 把 ft_min_word_len 变量设置为 3，重启服务器。如果你想让这个设置在服务器每次重启后都能生效，最好的办法是把这个设置放到某个选项文件里，如/etc/my.cnf 文件：

```
[mysqld]
ft_min_word_len=3
```

(2) 对于那些已经有 FULLTEXT 索引的现有数据表，你必须重新建立那些索引。你可以先删除、再重新创建，但更简便且同样有效的办法是进行一次快速修复操作：

```
REPAIR TABLE tbl_name QUICK;
```

(3) 在改变参数后创建的新 FULLTEXT 索引将自动使用新值。

关于如何设置系统变量的讨论见附录 D。使用选项文件的细节见附录 E。

说明 如果某个数据表有 FULLTEXT 索引，在使用 myisamchk 工具程序为该数据表重建索引的时候就必须注意一些与 FULLTEXT 索引有关的事项，详见附录 F 对 myisamchk 工具的描述。

第 3 章

数 据 类 型

数据库管理系统这个名字本身就意味着它是用来管理数据的，所以你在 MySQL 里做的每一件事情都或多或少地会涉及某种形式的数据。即使是一条简单得不能再简单的 SELECT 1 语句，也会涉及表达式的求值过程并且会产生一个整数类型的数据值。

MySQL 里的每一个数据值都有一个类型。比如说，37.4 是一个数值，'abc'是一个字符串。在某些场合，数据的类型是很直观的。比如说，当你用一条 CREATE TABLE 语句来创建一个数据表的时候，就必须明确地为这个数据表里的每一个数据列定义一种类型：

```
CREATE TABLE mytbl
(
  int_col   INT,        # integer-valued column
  str_col   CHAR(20),   # string-valued column
  date_col  DATE        # date-valued column
);
```

在另外一些场合，数据的类型就不那么直观了。比如说，当你在表达式里引用文本值、把值传递给一个函数、或者使用某个函数的返回值时：

```
INSERT INTO mytbl (int_col,str_col,date_col)
VALUES(14,CONCAT('a','b'),20090115);
```

这条语句将完成以下几个操作，每个操作都会涉及数据类型。

- ❑ 它把整数值 14 赋给一个整数类型的数据列 int_col。
- ❑ 它把字符串值'a'和'b'传递给 CONCAT()字符串拼接函数。CONCAT()函数返回字符串值'ab'，而这个值又被赋给一个字符串类型的数据列 str_col。
- ❑ 它把整数值 20090115 赋值给日期类型的数据列 date_col。在这个赋值操作中，出现了类型不匹配的问题。但整数值可以解释为数据类型，于是，MySQL 自动进行了一次自动类型转换，把整数值 20090115 转换为日期值'2002-01-15'。

要想使用好 MySQL，就必须透彻理解 MySQL 是如何处理数据的。本章的讨论重点有两个：一是 MySQL 所能处理的数据值的类型，二是这些类型在实际应用中需要注意的问题，如下所示。

- ❑ 包括 NULL 值在内，MySQL 能够表达哪几大类型的值？
- ❑ MySQL 为数据表里的数据列准备了哪些数据类型？每种数据类型都有哪些属性和特点？有些 MySQL 数据类型是相当常用的，如 BLOB 字符串类型。但有些类型（例如 TIMESTAMP 日期类型和有 AUTO_INCREMENT 属性的整数类型）却有着特殊的行为特点，不明白其中奥妙的人肯定会感到吃惊。

- 服务器的 SQL 模式如何影响坏数据的处理方式？严格模式会拒绝坏数据值。
- MySQL 的表达式求值规则。为方便对数据的检索、显示和计算处理，MySQL 为我们准备了很多可以用在表达式里的操作符和函数。类型转换规则是 MySQL 表达式求值规则的重要组成部分之一，它指的是当表达式需要用到某个类型的值而你提供的却是另外一种类型的值时，MySQL 将如何进行类型转换。而类型转换规则的要点则是类型转换会在何时发生以及到底如何进行。在某些场合，类型转换会导致不合理或者无意义的转换结果。比如说，当你把字符串`'13'`赋给一个整数类型的数据列时，MySQL 会自动把它转换为整数值 `13`。但当你把字符串`'abc'`赋给一个整数类型的数据列时，MySQL 的转换结果却是整数值 `0`（或产生错误），因为不存在与字符串`'abc'`对应的整数值。如果不熟悉 MySQL 的类型转换规则，你甚至会犯下更严重的错误。比如说，你本想只修改或者删除几个数据行，可结果却把整个数据表里的数据行全都修改或者删除掉了。
- 如何为数据列正确地选择数据类型。在创建表时如何挑选最佳的类型，以及当有好几种相关类型都适应于你要存储的值时，如何作出选择，知道这两点很重要。

作为本章关于 MySQL 数据类型、操作符和函数等讨论内容的补充，我在本书的附录 B 和附录 C 中还提供了一些额外的信息。

这一章的示例大量使用了 `CREATE TABLE` 和 `ALTER TABLE` 语句来创建和修改数据表。对于这条语句，相信读者不会感到陌生，因为我们在第 1 章和第 2 章已经用过它了。如有必要，请参考附录 E。

MySQL 支持好几种存储引擎，它们各有特点。在某些场合，一个给定数据类型的数据列对不同的存储引擎会有不同的行为，所以数据列的用法往往会对你选用哪一种存储引擎去创建数据表产生决定性的影响。本章只会偶尔涉及一些与存储引擎有关的问题，对各种存储引擎及其特点更详细的描述可以在第 2 章找到。

在某些场合，数据处理取决于当前的 SQL 模式和默认值是如何定义的。关于 SQL 模式的基础知识和设置办法，请参阅 2.1 节。就本章而言，3.2.3 节将讨论默认值的处理问题，3.3 节将讨论“严格”模式和针对“坏”数据的处理原则。

3.1 数据值的类别

不同类型的数据有着不同的表示形式。MySQL 能够识别和使用的数据值包括数值、字符串值、日期/时间值、坐标值和空值（`NULL`）。

3.1.1 数值

数值是指诸如 `48`、`193.62` 或`-2.378E12` 之类的值。MySQL 能够识别和使用的数值分为整数值（没有小数部分）、定点或浮点数值（可以有小数部分）、位字段值 3 种。

1. 精确值和近似值

MySQL 对精确值进行精确算术运算，对近似值进行近似运算。

精确值在运算过程中没有四舍五入之类的问题。精确值包括整数（如 `0`、`14`、`-382`）和带小数点的有理数（如 `0.1`、`38.5`、`-18.247`）。

整数既可以表示为十进制数字，也可以表示为十六进制数字。在十进制格式下，每个整数被表示为一个不包含小数点的数字序列。十六进制值被默认地当做字符串值来对待，但在进行数值运算的上

下文里，十六进制常数将被视为 64 位的整数。比如说，0x10 是十进制里的 16。3.1.2 节将对十六进制值的语法进行描述。

带小数部分的精确值由 3 部分组成：一个数字序列、一个小数点、再一个数字序列。小数点前后的两个数字序列中的一个可以为空，但它们不能同时为空。

近似值是用科学计数法表示的浮点数，它们带有一个底数和一个指数。具体做法是：在整数或浮点数的后面紧跟着字母 e 或 E，然后是一个正号或负号，然后是一个以整数表示的指数。底数和指数的正负号可以是任意可能的组合：1.58E5、-1.58E5、1.58E-5、-1.58E-5。

十六进制数不能用科学计数法来表示，这是因为科学计数法的指数部分必须以字母“e”开头，但“e”又是一个合法的十六进制数字，所以这样会产生二义性。

你可以在任何一个数值的前面加上一个正号或负号来表明它是一个正值或负值。

用精确值计算得到的结果是精确的，只要没有超出那些数值的精度范围，计算结果的准确性就不会受到任何影响。比如说，如果某个数据列只允许小数点后面有两位数字，就不能把 1.23456 原封不动地插入该数据列。对相近值的计算会很相近，而且容易发生舍入错误。

在对表达式进行求值的时候，MySQL 根据以下规则来决定应该进行精确计算还是近似计算。

- 只要表达式里有一个近似值，MySQL 将把它作为一个浮点（近似）表达式来求值。
- 如果表达式里的数值全都是整数形式的精确值，MySQL 将使用 BIGINT（64 位）精度对它进行求值。
- 如果表达式里的数值全是精确值、但其中有一个或多个值有小数部分，则使用 65 位精度进行 DECIMAL 运算。
- 如果在求值过程中必须把某个字符串转换为一个数值，它将被转换为一个双精度浮点值。于是，根据前面给出的规则，对该表达式的求值将是近似的。

2. 位字段值

位字段值（bit-field value）可以写成 b'val'或 0bval，其中 val 是由一个或多个二进制数字（0 或 1）构成的序列。比如说，b'1001'和 0b1001 是十进制里的 9。这些位值记号对应 MySQL 5.0.3 里的 BIT 数据类型，但位字段值在许多不同的上下文里都可以使用。

结果集里的 BIT 值将被显示为一个二进制字符串，这样的输出报告往往不容易阅读和理解。一个比较好的办法是先把它们转换为整数再输出，只要给它们分别加上一个零或使用 CAST()函数即可：

```
mysql> SELECT b'1001' + 0, CAST(b'1001' AS UNSIGNED);
+-------------+---------------------------+
| b'1001' + 0 | CAST(b'1001' AS UNSIGNED) |
+-------------+---------------------------+
|           9 |                         9 |
+-------------+---------------------------+
```

3.1.2　字符串值

字符串是诸如'Madison, Wisconsin'、'patient shows improvement'或'12345'（虽然它看起来像一个数值，但实际上并不是）之类的值。字符串值两端的引号既可以是单引号，也可以是双引号，但因为以下两个原因，应该尽量使用单引号。

- SQL 语言标准规定的是单引号，如果你在编写查询语句时使用的是单引号，就比较容易把它们移植到其他的数据库引擎去。

- 如果启用了 ANSI_QUOTES SQL 模式，MySQL 将把双引号里的字符串解释为一个标识符而不是把它解释为一个字符串，而这意味着双引号里的字符串必须是数据库或数据表的名字。请看下面这条语句：

```
SELECT "last_name" from president;
```

如果未启用 ANSI_QUOTES 模式，这条语句将从 president 数据表里对每行选取字符串 "last_name"。如果启用了 ANSI_QUOTES 模式，这条语句将从该数据表里选取 last_name 数据列的值。

接下来的例子使用了双引号来给出一个字符串（假设 ANSI_QUOTES 模式未启用）。

MySQL 能够识别出字符串里的几种转义序列，它们用来代表特殊字符，如表 3-1 所示。转义序列必须以一个反斜线字符（\）开始，而 MySQL 则会按转义规则来解释紧随其后的那个字符。需要提醒大家注意的是，NUL 字节与 SQL 语言中的 NULL 值是不一样的：NUL 代表的是零值字节，而 NULL 值代表的是“没有取值”的情况。

表3-1 字符串转义序列

转义序列	含　义
\0	NUL（零值字节）
\'	单引号
\"	双引号
\b	Backspace（后退）字符
\n	换行符
\r	回车符
\t	制表符
\\	反斜线符
\Z	Ctrl-Z（Windows系统中的EOF字符）

这个表里的转义序列是区分大小写的。对于那些没有列在这个表里的字符，即使在它们的前面加上一个反斜线字符，也仍将被解释为该字符本身。比如说，\t 是一个制表符，但\T 是一个普通的'T'字符。

从表 3-1 可以看出，你可以用反斜线序列来转义单引号或双引号，但这并不是唯一的办法。如果你的字符串里有引号（单引号或双引号）字符，可以用以下几种方法来表示。

- 如果字符串中的引号字符与括在字符串两端的引号相同，双写该引号。例如：

```
'I can''t'
"He said, ""I told you so."""
```

- 用与字符串中的引号字符不同的引号把该字符串括起来。此时，你用不着双写字符串中的引号字符，如下所示：

```
"I can't"
'He said, "I told you so."'
```

- 对字符串中的引号字符进行转义，这个办法不论字符串两端的引号是哪一种都可以使用，如下所示：

```
'I can\'t'
```

```
"I can\'t"
"He said, \"I told you so.\""
'He said, \"I told you so.\"'
```

如果需要取消反斜线字符的特殊含义而把它当做一个普通字符来对待，请启用 NO_BACKSLASH_SCAPE SQL 模式。

作为使用引号来写出字符串值的一个替代办法，你还可以使用两种十六进制表示法来写出字符串。第一种是使用标准化的 SQL 表示法 X'*val*'，其中 *val* 是几对十六进制数字（“0”到“9”和“a”到“f”）。比如说，X'0a'是十进制里的 10，X'ffff'是十进制里的 65535。前导字符“X”和字符串里的十六进制数字字符（“a”到“f”）以大写或小写字母的形式给出均可：

```
mysql> SELECT X'4A', x'4a';
+-------+-------+
| X'4A' | x'4a' |
+-------+-------+
| J     | J     |
+-------+-------+
```

在字符串上下文里，每两个十六进制数字被解释为一个 8 位的数值字节值，其取值范围是 0 到 255，而最终的结果将被用做一个字符串。在数值上下文里，十六进制常数都被当做一个数值来对待。下面的语句演示了十六进制常数在两种上下文里的解释情况：

```
mysql> SELECT X'61626364', X'61626364'+0;
+-------------+---------------+
| X'61626364' | X'61626364'+0 |
+-------------+---------------+
| abcd        |    1633837924 |
+-------------+---------------+
```

十六进制值还可以写成以“0x”开头、后面跟着一个或多个十六进制数字的序列。前缀“0x”区分大小写，所以“0x0a”和“0x0A”是合法的十六进制值，“0X0a”和“0X0A”不是。

类似于 X'*val*'的情况，0x 值被默认地解释为字符串，但在数值上下文里可以用做数值：

```
mysql> SELECT 0x61626364, 0x61626364+0;
+------------+--------------+
| 0x61626364 | 0x61626364+0 |
+------------+--------------+
| abcd       |   1633837924 |
+------------+--------------+
```

X'*val*'表示法要求构成 val 的数字的个数是偶数。像 X'a'这样的值是非法的。如果某个用 0x 表示法写出的值只有奇数个十六进制数字，MySQL 将给它增加一个前导的“0”字符。比如说，0xa 将被视为 0x0a。

1. 字符串类型与字符集支持

字符串值可以分为两大类：二进制和非二进制。

- 二进制字符串是一些字节序列。对这些字节的解释不牵涉任何字符集概念。二进制字符串没有特殊的比较或排序属性，比较操作将根据各个字节的数值逐字节地进行。所有字节都要参加比较，尾缀的空格也包括在内。
- 非二进制字符串是由字符构成的序列。每个非二进制字符串都与一个字符集相关联，字符集决定了哪些字符可以用来表示数据以及 MySQL 将如何解释字符串的内容。每种字符集都有一

种或多种排序方式。某给定字符串所使用的排序方式决定着字符集里的字符的先后顺序，这个顺序对比较操作的结果有着决定性的影响。默认的字符集和排序方式是 latin1 和 latin1_swedish-ci。

非二进制字符串里的尾缀空格不参加比较操作，但 TEXT 类型是个例外：基于索引的比较操作会在必要时在 TEXT 值的尾部补足一些空格，但如果你试图在一个取值唯一化的 TEXT 索引里插入一个新值，它与某个现有的值只在尾缀空格的数量方面有所差异，这就会导致"键字重复"错误。

不同的字符集对每个字符的存储空间要求是不同的。诸如 laltin1 之类的单字节字符集的每个字符只需要一个字节来存储，而多字节字符集的部分或全部字符则需要一个以上的字节来存储。比如说，MySQL 里的 Unicode 字符集就是多字节的。ucs2 是一个双字节字符集，它的每个字符需要两个字节来存储。utf8 是一个长度可变的多字节字符集，它的每个字符需要 1 到 3 个字节来存储。（从 MySQL 6.0.4 版本开始，utf8 字符最多需要占用 4 个字节。）

下面两条语句可以用来查知你的服务器都有哪些字符集和排序方式可供选用：

```
mysql> SHOW CHARACTER SET;
+----------+-----------------------------+---------------------+--------+
| Charset  | Description                 | Default collation   | Maxlen |
+----------+-----------------------------+---------------------+--------+
| big5     | Big5 Traditional Chinese    | big5_chinese_ci     |      2 |
| dec8     | DEC West European           | dec8_swedish_ci     |      1 |
| cp850    | DOS West European           | cp850_general_ci    |      1 |
| hp8      | HP West European            | hp8_english_ci      |      1 |
| koi8r    | KOI8-R Relcom Russian       | koi8r_general_ci    |      1 |
| latin1   | cp1252 West European        | latin1_swedish_ci   |      1 |
...
| utf8     | UTF-8 Unicode               | utf8_general_ci     |      3 |
| ucs2     | UCS-2 Unicode               | ucs2_general_ci     |      2 |
...
mysql> SHOW COLLATION;
+----------------------+----------+-----+---------+----------+---------+
| Collation            | Charset  | Id  | Default | Compiled | Sortlen |
+----------------------+----------+-----+---------+----------+---------+
| big5_chinese_ci      | big5     |   1 | Yes     | Yes      |       1 |
| big5_bin             | big5     |  84 |         | Yes      |       1 |
...
| latin1_german1_ci    | latin1   |   5 |         | Yes      |       1 |
| latin1_swedish_ci    | latin1   |   8 | Yes     | Yes      |       1 |
| latin1_danish_ci     | latin1   |  15 |         | Yes      |       1 |
| latin1_german2_ci    | latin1   |  31 |         | Yes      |       2 |
| latin1_bin           | latin1   |  47 |         | Yes      |       1 |
| latin1_general_ci    | latin1   |  48 |         | Yes      |       1 |
| latin1_general_cs    | latin1   |  49 |         | Yes      |       1 |
| latin1_spanish_ci    | latin1   |  94 |         | Yes      |       1 |
...
```

从 SHOW COLLATION 语句的输出报告里可看到的，每一种排序方式都与某个特定的字符集关联，而每一个给定的字符集可以有多种排序方式。排序方式的名字由字符集的名字、一种语言和一个额外的后缀构成。比如说，utf8_icelandic_ci 是 utf8 Unicode 字符集的排序方式之一，它使用冰岛（icelandic）排序规则来比较两个字符串，比较时区分字母的大小写情况。排序方式的名字里的后缀有

以下含义。

- _ci。这是一种不区分大小写的排序方式。
- _cs。这是一种区分大小写的排序方式。
- _bin。这是一种二进制排序方式。比较操作将根据字符的编码值来进行，不牵涉任何一种人类语言。因此，_bin 排序方式的名字里不包括任何人类语言的名字，如 latin1_bin 和 utf8_bin。

二进制字符串和非二进制字符串有着不同的排序特性，如下所示。

- 二进制字符串的比较操作是逐字节进行的，其结果只取决于各字节的数值大小。这个特性使得二进制字符串看起来像是区分大小写的（'abc' <> 'ABC'），这是因为同一个字符的大写和小写形式有着不同的数值字节值。二进制字符串其实没有大小写的概念。字母的大小写是一个与排序方式有关的概念，只适用于字符（非二进制）字符串。
- 非二进制字符串的比较操作是逐字符进行的，每个字符的相对值取决于当前所使用的字符集的排序方式。大多数排序方式把同一个字符的大写和小写版本设定为同样的排序值，所以非二进制字符串的比较操作基本上都是不区分大小写的。当然，这个结论对那些区分大小写的排序方式或者是二进制排序方式不成立。

排序方式决定着字符串比较和排序操作的结果，而这又会影响到许多与字符串有关的操作。

- 比较操作符。<、<=、=、<>、>=、>、LIKE。
- 排序操作。ORDER BY、MIN()、MAX()。
- 分组操作。GROUP BY、DISTINCT。

你可以用 CHARSET()或 COLLATION()函数来查出字符串所使用的字符集和排序方式。

MySQL 将根据当前的服务器设置来解释那些括在引号里的字符串值。默认的字符集和排序方式是 latin1 和 latin1_swedish-ci：

```
mysql> SELECT CHARSET('abcd'), COLLATION('abcd');
+-----------------+-------------------+
| CHARSET('abcd') | COLLATION('abcd') |
+-----------------+-------------------+
| latin1          | latin1_swedish_ci |
+-----------------+-------------------+
```

在默认的情况下，MySQL 把十六进制常数解释为二进制字符串：

```
mysql> SELECT CHARSET(X'0123'), COLLATION(X'0123');
+------------------+--------------------+
| CHARSET(X'0123') | COLLATION(X'0123') |
+------------------+--------------------+
| binary           | binary             |
+------------------+--------------------+
```

有两种表示法可以让 MySQL 强行使用一种给定的字符集来解释字符串值。首先，如下所示的表示法可以让 MySQL 强行使用一种给定的字符集来解释一个字符串常量，其中的 charset 是当前可用的一种字符集的名字：

```
_charset str
```

_charset 表示法被称为字符集引导（character set introducer)，它修饰的字符串既可以在引号里写出，也可以是一个十六进制值。下面的例子演示了如何用 latin2 和 utf8 字符集来解释字符串：

```
_latin2 'abc'
_latin2 X'616263'
_latin2 0x616263
_utf8 'def'
_utf8 X'646566'
_utf8 0x646566
```

对于引号里的字符串，字符集引导和字符串之间的空白是可选的。对于以十六进制值的形式给出的字符串，这个空白是必不可少的。

3

其次，表示法 N'str'等价于_utf8 'str'。注意，“N”（大小写不敏感）的后面必须紧跟着括在引号里给出的字符串，它们之间不能有任何空白。

加上一个字符集引导的办法只适用于括在引号里给出的字符串或十六进制常量，不适用于字符串表达式或数据列里的值。CONVERT()函数可以把任意一个字符串值转换为一个按某给定字符集来解释的字符串：

```
CONVERT(str USING charset);
```

字符集引导和 CONVERT()函数是不一样的。字符集限定符只改变对字符串的解释，不改变字符串值本身(对于多字节字符集,它可能会在字符串包含的字节数不足时增加一些尾缀的空格)。CONVERT()函数是把给定的字符串作为一个输入参数，然后根据给定的字符集生成一个新的字符串。

字符集引导和 CONVERT()函数的区别可以从下面这个例子里看出来，例子里使用了 ucs2 双字节字符集：

```
mysql> SET @s1 = _ucs2 'ABCD';
mysql> SET @s2 = CONVERT('ABCD' USING ucs2);
```

假设默认的字符集是 latin1（一个单字节字符集）。第一条语句将把字符串'ABCD'中的每两个字解释为一个双字节的 ucs2 字符，结果是一个两个字符的 ucs2 字符串。第二条语句将把字符串'ABCD'中的每个字符转换为相应的 ucs2 字符，结果是一个 4 个字符的 ucs2 字符串。

字符串的“长度”是什么？这要视情况而定。如果用 CHAR_LENGTH()函数去测量，长度是字符的个数。如果用 LENGTH()函数去测量，长度是字节的个数。对一个包含多字节字符的字符串来说，这两个值是不一样的：

```
mysql> SELECT CHAR_LENGTH(@s1), LENGTH(@s1), CHAR_LENGTH(@s2), LENGTH(@s2);
+------------------+-------------+------------------+-------------+
| CHAR_LENGTH(@s1) | LENGTH(@s1) | CHAR_LENGTH(@s2) | LENGTH(@s2) |
+------------------+-------------+------------------+-------------+
|                2 |           4 |                4 |           8 |
+------------------+-------------+------------------+-------------+
```

还有一个更容易让人犯糊涂的问题：一个二进制字符串和一个使用某种二进制排序方式的非二进制字符串是不一样的，如下所述。

- ❑ 二进制字符串不涉及字符集的概念。它按字节来解释，比较操作逐个字节地比较各个字节的数值代码。
- ❑ 非二进制字符串即便是使用二进制排序方式，也要按字符来解释，比较操作逐个字符地比较各个字符的数值代码，每个字符可能涉及多个字节。

下面这个例子可以让我们看到二进制字符串和非二进制字符串在大小写方面的区别。先创建一个二进制字符串和一个使用了某种二进制排序方式的非二进制字符串，然后把它们传递给 UPPER()

函数：

```
mysql> SET @s1 = BINARY 'abcd';
mysql> SET @s2 = _latin1 'abcd' COLLATE latin1_bin;
mysql> SELECT UPPER(@s1), UPPER(@s2);
+------------+------------+
| UPPER(@s1) | UPPER(@s2) |
+------------+------------+
| abcd       | ABCD       |
+------------+------------+
```

UPPER()函数为什么没能把二进制字符串转换为大写？这是因为二进制字符串没有任何字符集，所以无从得知哪些字节值对应着大写或小写字符。如果需要把一个二进制字符串传递给诸如 UPPER()或 LOWER()之类的函数，必须先把它转换为非二进制字符串：

```
mysql> SELECT @s1, UPPER(CONVERT(@s1 USING latin1));
+------+----------------------------------+
| @s1  | UPPER(CONVERT(@s1 USING latin1)) |
+------+----------------------------------+
| abcd | ABCD                             |
+------+----------------------------------+
```

2. 与字符集有关的系统变量

MySQL 服务器通过几个系统变量来调控对字符集的支持情况。这些变量中的大多数与字符集有关，其他的与排序方式有关。每一个排序方式变量都与一个相应的字符集变量相关联。

有些字符集变量用来表明服务器或当前数据库的属性，如下所示。

- character_set_system。各种标识符所使用的字符集。这永远是 utf8。
- character_set_server 和 collation_server。服务器的默认字符集和排序方式。
- character_set_database 和 collation_database。默认数据库的字符集和排序方式。这两个变量是只读的，服务器会在你选择一个默认的数据库时自动地设置好它们。如果没有默认的数据库，它们将被设置为服务器的默认字符集和排序方式。如果你在创建某个数据表时没有明确地为之指定字符集和排序方式，MySQL 就会使用这些变量的设置值来创建它——数据表的默认设置将沿用数据库的默认设置。

其他的字符集变量将影响客户和服务器之间通信，如下所示。

- character_set_client。客户向服务器发送 SQL 语句时使用的字符集。
- character_set_results。服务器向客户返回结果时使用的字符集。这里所说的“结果”包括数据值和诸如数据列名字之类的元数据。
- character_set_connection。一个由服务器使用的变量。当服务器接收到来自客户的语句字符串时，它将把该字符串从 character_set_client 转换为 character_set_connection，并使用后者对该语句进行处理。（这里有一个例外：如果在语句里有带着字符集限定符的文本字符串值，MySQL 将使用由字符集限定符给定的字符集来解释有关的字符串。）collation_set_connection 也是 MySQL 对 SQL 语句字符串里的两个文本字符串值进行比较时使用的字符集。
- character_set_filesystem。文件系统使用的字符集。这个字符集用来解释 SQL 语句（比如 LOAD DATA 语句）里的代表着文件名的文本字符串值。这些文件名字符串将先从 character_

set_client 转换为 character_set_filesystem，然后再被用来打开文件。这个变量的默认值是 binary（不必转换）。

在默认的情况下，绝大多数字符集和排序方式变量被设置成相同的值。比如说，如下所示的输出报告表明：客户和服务器之间的通信使用的是 latin1 字符集。

```
mysql> SHOW VARIABLES LIKE 'character\_set\_%';
+--------------------------+--------+
| Variable_name            | Value  |
+--------------------------+--------+
| character_set_client     | latin1 |
| character_set_connection | latin1 |
| character_set_database   | latin1 |
| character_set_filesystem | binary |
| character_set_results    | latin1 |
| character_set_server     | latin1 |
| character_set_system     | utf8   |
+--------------------------+--------+
mysql> SHOW VARIABLES LIKE 'collation\_%';
+----------------------+-------------------+
| Variable_name        | Value             |
+----------------------+-------------------+
| collation_connection | latin1_swedish_ci |
| collation_database   | latin1_swedish_ci |
| collation_server     | latin1_swedish_ci |
+----------------------+-------------------+
```

如果某个客户想使用另一种字符集与服务器通信，可以修改与通信有关的变量。比如说，如果你想使用 utf8 字符集与服务器通信，需要改变 3 个变量：

```
mysql> SET character_set_client = utf8;
mysql> SET character_set_results = utf8;
mysql> SET character_set_connection = utf8;
```

不过，用一条 SET NAMES 语句来完成这种修改更方便。下面这条语句的效果相当于上面 3 条 SET 语句：

```
mysql> SET NAMES 'utf8';
```

在修改与通信有关的字符集变量时有一个限制：不能使用 ucs2 字符集。（在 MySQL 6.0 和更高的版本里，也不能使用 utf16 和 utf32 字符集。）

有许多客户程序支持一个--default-character-set 选项，这个选项的使用效果和 SET NAMES 语句相同：告诉 MySQL 服务器你想使用另一种字符集与之通信。

对于那些成对出现的变量（一个字符集变量和一个排序方式变量），两个变量之间通过以下方式相互关联。

- ❑ 改变字符集变量将把相关的排序方式变量设置为新字符集的默认排序方式。
- ❑ 改变排序方式变量将把相关的字符集变量设置为新排序方式的名字的第一部分所给出的字符集。

比如说，把 character_set_connection 变量设置为 utf8 将把 collation_connection 变量设置为 utf8_general_ci。把 collation_connection 变量设置为 latin1_spanish_ci 将把 character_set_connection 变量设置为 latin1。

3.1.3 日期/时间值

日期/时间值指的是'2011-06-17'或'12:30:43'这样的值。MySQL 允许把日期和时间合并在一起来表示，如'2011-06-17 12:30:43'。有一点请特别注意：MySQL 是按“年-月-日”的顺序来表示日期的。虽说这是 SQL 标准所规定的格式（也叫做“ISO 8601 格式”），但很多 MySQL 初学者还是会感到不习惯。利用 DATE_FORMAT()函数，你可以按任意顺序来显示日期值，但“年-月-日”的顺序是 MySQL 默认的日期显示格式。而且，当你输入日期值时，也必须按“年-月-日”的顺序进行。对于其他格式的值，你可能需要先用 STR_TO_DATE()函数对它们进行转换后才能输入。

3.1.4 坐标值

MySQL 支持坐标值，但仅限于 MyISAM 数据表以及（从 MySQL 5.0.16 版本开始）InnoDB、NDB 和 ARCHIVE。这使我们可以把用来表示点、线、多边形的值存放到 MySQL 数据库里。比如说，下面的语句使用 X、Y 坐标值（10, 20）创建了一个 POINT 类型的值，并把这个值的文本表示形式赋值给了一个用户定义的变量：

```
SET @pt = POINTFROMTEXT('POINT(10 20)');
```

3.1.5 布尔值

在表达式里，零表示“假”（false），而任何非零、非 NULL 的值都表示“真”（true）。

布尔常数 TRUE 和 FALSE 在表达式里将被分别求值为 1 和 0。它们是不区分大小写的。

3.1.6 空值NULL

NULL 是一个“没有类型”的值。它通常用来表示“没有数据”、“数据未知”、“数据缺失”、“数据超出取值范围”、“对本数据列不适用”、“与本数据列的其他值不同”等含义。你可以把 NULL 值插入数据表，可以从数据表检索它们，可以测试某个值是不是 NULL。但你不能对 NULL 进行数学运算，如果你一意孤行，则计算结果将永远是 NULL。此外，有许多函数会在你使用 NULL 值或非法参数调用它们时返回 NULL。

在写 NULL 的时候，不需要使用引号，也不必区分字母的大小写情况。MySQL 还把单独的“\N”（注意，必须是大写的 N）解释为 NULL：

```
mysql> SELECT \N, ISNULL(\N);
+------+------------+
| NULL | ISNULL(\N) |
+------+------------+
| NULL |          1 |
+------+------------+
```

3.2 MySQL 的数据类型

数据库里的数据表是由一个或者多个数据列构成的。在用 CREATE TABLE 语句创建数据表时，你必须为它的每个数据列定义一个类型。数据列类型要比一般的分类（如“数值”或“字符串”等）更细致。数据列类型是对“数据列里的值有哪些具体特性”这个问题的准确回答，如 SMALLINT 或

VARCHAR(32)等。数据列类型决定了 MySQL 将如何对待那些值。例如，你既可以把数值类型的值保存在一个数值类型的数据列里，也可以把它们保存在一个字符串类型的数据列里，MySQL 会根据你为这些值选择的存储方式而区别对待这两种情况。每一种列类型都有以下几种属性（或者说可以回答以下一些问题）。

- ❑ 你可以把哪些种类的值保存在其中？
- ❑ 这种类型的值要占用多少存储空间？
- ❑ 值的长度是固定不变的（即该类型的每个值都将占用同样数量的存储空间）还是可变的（不同的值会占用不同的存储空间）？
- ❑ 这种类型的值如何进行比较和存储？
- ❑ 能否对这种类型编制索引？

下面的讨论先简要介绍 MySQL 的各种数据类型，再详细描述它们的定义语法和每种类型的特性，如取值范围和内存占用量。我们将按照各数据类型在 CREATE TABLE 语句里的用法来描述其特征。语法中的方括号部分是可选信息。比如说，语法 MEDIUMINT [(*M*)]表明该类型的最大显示宽度(*M*)是可选的。反过来说，因为语法 VARCHAR (*M*)中没有方括号，所以这里的(*M*)是必不可少的。

3.2.1 数据类型概述

MySQL 的数值数据类型包括整数、定点数、浮点数和位值，如表 3-2 所示。除 BIT 之外的数值类型既可以带正负符号，也可以不带正负符号。如果想让数据列自动生成一组有序的整数或浮点值，可以使用一种特殊的属性，这特别适用于你需要一组独一无二的标识编号的情况。

表3-2 数值数据类型

数据类型	含　义
TINYINT	非常小的整数
SMALLINT	小整数
MEDIUMINT	中等大小的整数
INT	标准的整数
BIGINT	大整数
DECIMAL	定点数
FLOAT	单精度浮点数
DOUBLE	双精度浮点数
BIT	位字段

表 3-3 给出了 MySQL 的字符串类数据类型。字符串可以容纳任何东西，包括图像和声音等各种二进制数据。字符串可以彼此比较，而比较操作又分为区分字母大小写和不区分字母大小写两种情况。此外，你还可以对字符串进行模式匹配。（事实上，MySQL 的数值类型也允许进行模式匹配，只不过字符串数据类型上的模式匹配操作更多见一些。）

表 3-4 给出了 MySQL 的日期/时间数据类型，表中的 *CC*、*YY*、*MM*、*DD*、*hh*、*mm* 和 *ss* 分别代表世纪、年、月、日、时、分、秒。MySQL 提供的时间类列类型有：日期与时间（合并表示或分开表示）、时间戳（一种专门用来记载某数据记录最近一次修改时间的类型）。此外，为了方便不需要完整日期而只需要年份数值的场合，MySQL 还准备了一个 YEAR 类型。

表3-3　字符串数据类型

数据类型	含　　义
CHAR	固定长度的非二进制（字符）字符串
VARCHAR	可变长度的非二进制字符串
BINARY	固定长度的二进制字符串
VARBINARY	可变长度的二进制字符串
TINYBLOB	非常小的BLOB（二进制大对象）
BLOB	BLOB
MEDIUMBLOB	中等大小的BLOB
LONGBLOB	大BLOB
TINYTEXT	非常小的非二进制字符串
TEXT	小文本字符串
MEDIUMTEXT	中等大小的非二进制字符串
LONGTEXT	大的非二进制字符串
ENUM	枚举集合。数据列的取值将是这个枚举集合中的某一个元素
SET	集合。数据列的取值可以是零或者这个集合中的多个元素

表3-4　日期/时间数据类型

数据类型	含　　义
DATE	日期值，格式为 '*CCYY-MM-DD*'
TIME	时间值，格式为 '*hh:mm:ss*'
DATETIME	日期加时间值，格式为'*CCYY-MM-DD hh:mm:ss*'
TIMESTAMP	时间戳值，格式为'*CCYY-MM-DD hh:mm:ss*'
YEAR	年份值，格式为 *CCYY*或*YY*

表 3-5 展示了 MySQL 空间数据类型。这里展现了各种几何值和位置值。

表3-5　空间数据类型

数据类型	含　　义
GEOMETRY	任何类型的坐标值
POINT	点（一对X、Y坐标值）
LINESTRING	曲线（一个或多个POINT值）
POLYGON	多边形
GEOMETRYCOLLECTION	GEOMETRY值的集合
MULTILINESTRING	LINESTRING值的集合
MULTIPOINT	POINT值的集合
MULTIPOLYGON	POLYGON值的集合

3.2.2　数据表中的特殊列类型

在用 CREATE TABLE 语句创建数据表时，必须把构成该数据表的全体数据列都写出来。下面这个示例创建一个名为 mytbl 的数据表，该数据表由 3 个数据列构成，分别叫做 f、c 和 i：

```
CREATE TABLE mytbl
(
  f FLOAT(10,4),
  c CHAR(15) NOT NULL DEFAULT 'none',
  i TINYINT UNSIGNED NULL
);
```

每一个数据列都有一个名字和一个类型，而每一种类型又都关联有一些属性。数据列的声明语法如下所示：

col_name col_type [*type_attributes*] [*general_attributes*]

col_name 是该数据列的名字，它永远出现在数据列定义的开头，并且必须是合法的标识符。标识符的命名规则已经在 2.2 节介绍过了。简单地说，数据列标识符可以多达 64 个字符，允许用在数据列名字里的字符包括 MySQL 服务器的默认字符集中的字母和数字以及下划线（_）和美元（$）符号。关键字（如 SELECT、DELETE、CREATE 等）通常保留，不得用做数据列的名字。不过，数据列名字允许包含其他字符，也允许把保留字用做数据列的名字，但前提是必须把这个名字用引号（'）括起来。如果启用了 ANSI_QUOTES 模式，可以选择用双引号（"）引用标识符。

col_type 给出了数据列类型，即表明这个数据列可以用来容纳什么样的数据值。某些类型需要用说明符指定保持在数据列中的值的最大长度，另外一些类型的长度限制则隐含在它们的名称里。比如说，CHAR(10)将把数据列的长度明确地设定为 10 个字符，而 TINYBLOB 则隐含地把数据列的最大长度设置为 255 个字符。有些类型说明符允许你设定一个最大显示宽度（要用多少个字符来显示有关的数据值）。对于定点类型和浮点类型，你还可以设定有效位数和小数位。

在 col_type 的后面，我们可以设置可选的、特定于类型的属性，以及普通的。这些属性的作用是对该类型做进一步的修饰和限定，会对 MySQL 处理这个数据列的方式产生相应的影响，如下所示。

- 不同的数据列类型有着不同的属性。比如说，只有数值类型才有 UNSIGNED 和 ZEROFILL 属性，只有非二进制文本类型才有 CHARACTER SET 和 COLLATE 属性。
- 有些属性几乎可以用来修饰任何一种数据类型。比如说，你可以用 NULL 或 NOT NULL 来限定某个数据列是否允许容纳 NULL 值。对于大多数数据类型，都可以通过 DEFAULT 子句来为数据列定义默认值。3.2.3 节将介绍默认的值处理方式。

如果需要为数据列设定多项属性，它们的先后顺序就必须遵守一定的规则。一般说来，为减少不必要的麻烦，你最好把某数据类型的唯一属性（如 UNSIGNED 或 ZEROFILL）安排在它通用属性（如 NULL 和 NOT NULL）的前面。

3.2.3　指定列默认值

除 BLOB/TEXT 类型、空间类型以及具有 AUTO_INCREMENT 属性的数据列以外，你可以用 DEFAULT *def_value* 子句为某给定数据列设定一个默认值，如果在创建一个新数据行时没有为该数据列明确地给出一个值，该数据列将被赋值为这个默认值。*def_value* 必须是一个常数（但 *TIMESTAMP* 数据列有几种例外情况），它不能是一个表达式，也不能引用其他数据列。

如果某个数据列定义没有明确地包括 DEFAULT 子句、但该数据列允许为 NULL 值，它的默认值将是 NULL。在此基础上，对缺少 DEFAULT 子句的处理因 MySQL 的版本而异。

从 MySQL 5.0.2 版开始，数据列在创建时没有任何 DEFAULT 子句。也就是说，它没有任何默认值。这对服务器将如何处理新数据行——如果在插入该数据行时没有给出某个数据列的值的话——造成

了以下影响。

- 如果没有启用严格 SQL 模式，该数据列将被设置为它的数据类型的隐含默认值（各种隐含默认值将在稍后讨论）。
- 如果启用了严格 SQL 模式，支持事务的数据表将报告一个错误，该语句将执行失败，本次事务将回滚。对于不支持事务的数据表，如果该语句插入的新数据行是数据表里的第一个数据行，该语句将执行失败并报告一个错误。如果它不是第一个数据行，MySQL 将有两种选择：一是让该语句执行失败，二是把那个数据列设置为它的隐含默认值并发出一条警告消息。具体选择何种做法取决于启用了哪种严格模式设置，详见 3.3 节。

在 MySQL 5.0.2 版以前，MySQL 在定义数据列时要求使用一个 DEFAULT 子句来为它定义一个隐含的默认值。

数据列的隐含默认值取决于它的数据类型。

- 对于数值类型的数据列（不包括那些具有 AUTO_INCREMENT 属性的数据列），默认值是 0。对于 AUTO_INCREMENT 数据列，默认值是下一个可用序号。
- 对于日期和时间类型的数据列（不包括 TIMESTAMP 数据列），默认值是该数据类型的“零值”（比如说，DATE 类型的零值是'0000-00-00'）。TIMESTAMP 数据列的情况分为两种：如果它是数据表里的第一个 TIMESTAMP 数据列，默认值是当前日期和时间；如果它不是数据表里的第一个 TIMESTAMP 数据列，默认值是“零值”。（TIMESTAMP 数据列的默认值问题其实要比这复杂得多，详见 3.2.6 节中的第 2 小节。）
- 对于字符串类型（不包括 ENUM 类型）的数据列，默认值是空字符串。对于 ENUM 数据列，默认值是枚举集合里的第一个元素。对于 SET 数据列，如果不允许包含 NULL 值，默认值将是一个空集合，但它等价于一个空字符串。

你可以用 SHOW CREATE TABLE 语句来查看哪些数据列有 DEFAULT 子句以及它们的默认值到底是什么。

3.2.4 数值数据类型

MySQL 的数值数据类型分为 3 大类，如下所示。

- 精确值类型，它包括整数类型和 DECIMAL。整数类型用来存放没有小数部分的数值，如 43、−3、0、−798432 等。只要是能够用整数来表示的数据（如精确到磅的重量值、精确到英寸的长度值、家庭人口数、银河中的恒星数、培养皿中的细菌数等），都可以用一个整数类型的数据列来保存。DECIMAL 类型保存的精确值可以有一个小数部分，如 3.14159、−.00273、−4.78 等。这种数据类型非常适合用来保存财务金额数据。只要有可能，进入数据库的整数和 DECIMAL 值就会和你录入的完全一样，不存在四舍五入的问题，用它们进行的计算也是精确的。
- 浮点类型，它细分为单精度（FLOAT）和双精度（DOUBLE）。这些类型和 DECIMAL 类型一样，也可以用来存放有小数部分的数值，但它们容纳的是可能发生四舍五入的近似值，如 3.9E+4 或−0.1E−100。如果对数值精确度的要求不那么严格或是数值大到 DECIMAL 类型无法表示，浮点类型会是不错的选择。诸如粮食平均亩产、空间距离、失业率之类的数据都很适合用浮点值来保存。
- BIT 类型用来保存位字段值（bit-field value）。

可以把带小数部分的值放到一个整数类型的数据列里去，但它将被四舍五入为一个整数：如果小

数部分大于或等于 0.5，舍弃小数部分，整数部分加上 1（负数是减去 1）。类似地，你也可以把整数值放到一个浮点类型的数据列里去，它将被看做是一个小数部分等于零的浮点数。

在给出数值时，不应该使用逗号作为分隔符。比如说，`12345678.90` 是合法的，`12,345,678.90` 是非法的。

表 3-6 列出了各种数值类型的名称和取值范围，表 3-7 列出了各种数值类型的存储空间要求。*M* 表示整数类型的最大显示宽度、浮点类型和 DECIMAL 类型的精度（小数点后面的数字）以及 BIT 类型的位数。对于那些有小数部分的数据类型，*D* 代表数学精确度（小数点后面的数字个数），也称为“小数精度”（scale）。

3

表3-6　数值数据类型的取值范围

类型定义	取值范围
TINYINT [(*M*)]	带符号值：−128到127（-2^7到2^7-1）
	无符号值：0到255（0到2^8-1）
SMALLINT [(*M*)]	带符号值：−32768到32767（-2^{15}到$2^{15}-1$）
	无符号值：0到65535（0到$2^{16}-1$）
MEDIUMINT [(*M*)]	带符号值：−8388608到8388607（-2^{23}到$2^{23}-1$）
	无符号值：0到16777215（0到$2^{24}-1$）
INT [(*M*)]	带符号值：−2147483648到2147483647（-2^{31}到$2^{31}-1$）
	无符号值：0到4294967295（0到$2^{32}-1$）
BIGINT [(*M*)]	带符号值：−9223372036854775808到9223372036854775807（-2^{63}到$2^{63}-1$）
	无符号值：0到18446744073709551615（0到$2^{64}-1$）
DECIMAL ([*M* [, *D*]])	可变；取值范围由*M*和*D*的值决定
FLOAT [(*M*, *D*)]	最小非零值：±1.175494351E−38
	最大非零值：±3.402823466E+38
DOUBLE [(*M*, *D*)]	最小非零值：±2.2250738585072014E−308
	最大非零值：±1.7976931348623157E+308
BIT[(*M*)]	0到2^M-1

表3-7　数值数据类型的存储空间要求

类型定义	存储空间占用量
TINYINT [(*M*)]	1字节
SMALLINT [(*M*)]	2字节
MEDIUMINT [(*M*)]	3字节
INT [(*M*)]	4字节
BIGINT [(*M*)]	8字节
DECIMAL ([*M* [*D*]])	可变，取决于*M*和*D*
FLOAT [(*M*, *D*)]	4字节
DOUBLE [(*M*, *D*)]	8字节
BIT [(*M*)]	可变，取决于*M*

DECIMAL 值的存储空间要求取决于小数点左右两侧的数字的个数。对于每一侧，每 9 位数字需要 4 个字节，最后剩下的数字需要 1 到 4 个字节。每个 DECIMAL 值需要的存储空间是小数点左右两侧数值所需要的存储空间的总和。

一个 BIT(*M*)值需要大约(*M*+7)/8 个字节的存储空间。

1. 精确值数值数据类型

精确值数据类型包括整数类型和定点 DECIMAL 类型。

MySQL 中的整数类型包括 TINYINT、SMALLINT、MEDIUMINT、INT 和 BIGINT。INTEGER 是 INT 的一个同义词。这几种数据类型的主要区别在于它们所代表的值的取值范围和它们所需要的存储空间各不相同。（取值范围越大，需要的存储空间就越多。）整数数据列可以被定义为 UNSIGNED 以表明不允许使用负数，这将把取值范围上移到从 0 开始的区间。

在定义整数列时，你可以给它指定一个可选的显示宽度 *M*。指定的 M 值必须是 1 到 255 之间的一个整数。它决定着 MySQL 将用多少个字符来显示该数据列里的值。比如说，MEDIUMINT(4)定义了一个 MEDIUMINT 数据列，这个数据列的显示宽度是 4 个字符。如果你没有给整数列声明一个显式宽度，MySQL 将自行确定一个默认的宽度，这个默认宽度通常是该整数列里“最长”的值的长度。需要特别注意的是，数据列里的值并不会因为你设置了显示宽度而被截短为 *M* 个字符。如果某个值需要有 M 个以上的字符才能打印出来，MySQL 会把它完整地显示出来。

整数列的显示宽度 *M* 指的是 MySQL 将用多少个字符来显示该数据列里的值，与整数值需要用多少个字节来存储毫不相干。比如说，不管显示宽度是多少个字符，BIGINT 值都要占用 8 个字节的存储空间。即使你把它写成 BIGINT(4)，BIGINT 数据列所要求的存储空间也不可能奇迹般地缩小一半。这个 *M* 与数据列的取值范围也没有任何关系，你可以把某个数据列声明为 INT(3)，但这并不会把这个数据列里的最大值限定为 999。

DECIMAL 是一种定点类型，构成 DECIMAL 值的十进制数字的个数是固定的。这个事实的最大优点是 DECIMAL 值不像浮点数那样存在四舍五入的问题——这个特点使得 DECIMAL 类型非常适合用来保存财务数据。

NUMERIC 和 FIXED 都是 DECIMAL 的同义词。

DECIMAL 数据列也可以被定义成 UNSIGNED。与整数类型不同的是，把一个 DECIMAL 类型的数据列定义为 UNSIGNED 不会扩大该数据列的取值范围，其效果只是“砍掉”整个负数部分而已。

对于 DECIMAL 数据列，你可以给出一个有效数字的最大个数 *M* 和一个小数部分数字个数 *D*。它们分别对应于“精度”和“小数位数”概念，这些概念应该是你比较熟悉的。*M* 的取值范围是从 1 到 65，*D* 的取值范围是从 0 到 30 且不得大于 *M*。

M 和 *D* 都是可选的。如果省略了 *D*，它的默认值将是 0。如果省略了 *M*，它的默认值将是 10。换句话说，以下等效关系是成立的：

```
DECIMAL = DECIMAL(10) = DECIMAL(10,0)
DECIMAL(n) = DECIMAL(n,0)
```

DECIMAL 类型的取值范围要取决于 *M* 和 *D* 的值。如果在保持 *D* 不变的情况下调整 *M* 的值，取值范围将随着 *M* 的变大而增加（表 3-8）。如果在保持 *M* 不变的情况下调整 *D* 的值，取值范围将随着 *D* 的变大而减小（表 3-9）。

表3-8 *M*对**DECIMAL(*M*, *D*)**的取值范围的影响

类型定义	取值范围
DECIMAL (4, 1)	−999.9到999.9
DECIMAL (5, 1)	−9999.9到9999.9
DECIMAL (6, 1)	−99999.9到99999.9

表3-9 *D*对**DECIMAL(*M*, *D*)**的取值范围的影响

类型定义	取值范围
DECIMAL (4, 0)	−9999到9999
DECIMAL (4, 1)	−999.9到999.9
DECIMAL (4, 2)	−99.99到99.99

说明 MySQL 5.0.3 以前的版本把 DECIMAL 值存储为字符串，因而有一些与它们现在的表示方法不同的属性。详细情况请参阅《MySQL 参考手册》。如果你需要把一个老数据表里的 DECIMAL 数据列转换为最新的格式，可以先用 mysqldump 工具程序备份那个老数据表，再重新加载备份文件：

```
% mysqldump db_name tbl_name > file_name
% mysql db_name < file_name
```

2. 近似值数值数据类型

MySQL 提供了 FLOAT 和 DOUBLE 两种浮点类型来保存近似值。DOUBLE PRECISION 是 DOUBLE 的同义词。和 RAEL 在默认的情况下是 DOUBLE 的同义词。如果启用了 REAL_AS_DEFAULT SQL 模式，REAL 将变成 FLOAT 的同义词。

你可以给浮点类型加上 UNSIGNED 属性声明，这将把浮点类型的负数部分“砍掉”。

你可以给浮点类型设定一个有效数字的最大个数 *M* 和一个小数部分数字个数 *D*（就像对待 DECIMAL 类型那样）。*M* 是一个 1 到 255 之间的整数值，*D* 是一个 0 到 30 之间的整数值且不能大于 M。

对于 FLOAT 和 DOUBLE 类型，*M* 和 *D* 都是可选的。如果在定义数据列时省略了它们，MySQL 将按你的硬件所能允许的最大精度来存储该数据列的值。

MySQL 还允许使用 FLOAT(*p*)形式的声明语法。*p* 在 SQL 语言标准里表示某给定精度的数值所需要的位数，但 MySQL 对 *p* 会做出不同的解释：*p* 的取值范围是从 0 到 53，它只被用来确定某给定数据列将要存储的是单精度值还是双精度值。如果 *p* 值落在 0 到 24 的区间内，数据列将被视为单精度；如果 *p* 值落在 25 到 53 的区间内，数据列将被视为双精度。从效果上看，用 FLOAT(*p*)语法定义出来的数据列将根据 *p* 值的大小而被当做一个 FLOAT 数据列或一个 DOUBLE 数据列来对待，其 *M* 和 *D* 都使用默认值。

3. BIT数据类型

BIT 数据类型始见于 MySQL 5.0.3 版本，它是一种专门用来保存位字段值的类型。在定义一个 BIT 数据列时，你可以设定一个可选的最大宽度 *M*，它表示以位计算的数据列的宽度。*M* 的取值必须是 1 到 64 之间的一个整数。如果被省略，*M* 的默认值将是 1。

在默认的情况下，从 BIT 数据列检索出来的值不能显示为可打印形式。如果需要把一个位字段值

显示为可打印的表示形式，办法是给它加上零或是使用 CAST()函数：

```
mysql> CREATE TABLE t (b BIT(3));  # 3-bit column; holds values 0 to 7
mysql> INSERT INTO t (b) VALUES(0),(b'11'),(b'101'),(b'111');
mysql> SELECT b+0, CAST(b AS UNSIGNED) FROM t;
+------+---------------------+
| b+0  | CAST(b AS UNSIGNED) |
+------+---------------------+
|    0 |                   0 |
|    3 |                   3 |
|    5 |                   5 |
|    7 |                   7 |
+------+---------------------+
```

如果需要以二进制表示法来显示一些位字段值或是它们的计算结果，BIN()函数可以帮上大忙：

```
mysql> SELECT BIN(b), BIN(b & b'101'), BIN(b | b'101') FROM t;
+--------+-----------------+-----------------+
| BIN(b) | BIN(b & b'101') | BIN(b | b'101') |
+--------+-----------------+-----------------+
| 0      | 0               | 101             |
| 11     | 1               | 111             |
| 101    | 101             | 101             |
| 111    | 101             | 111             |
+--------+-----------------+-----------------+
```

4. 数值数据类型的属性

UNSIGNED 属性不允许数据列里出现负数值。它可以用来“修饰”除 BIT 以外的所有数值类型，但最常见的是与各种整数类型一起使用。给一个整数数据列加上 UNSIGNED 属性并不会改变该数据列的取值范围的“长度”，它只是把范围朝正数方向平移了，使其下限值从 0 开始而已。请看下面这个数据表声明：

```
CREATE TABLE mytbl
(
  itiny   TINYINT,
  itiny_u TINYINT UNSIGNED
);
```

itiny 和 itiny_u 都是 TINYINT 数据列，它们取值范围的“长度”都是 256，但具体的可取值不一样。itiny 的取值范围是-128 到 127，而 itiny_u 的取值范围上移到了 0 到 255。

如果数据不可能出现负数值，比如人口统计数字或观众人数等情况，就应考虑给相应的整数数据列加上 UNSIGNED 属性。如果用一个带正负号的数据列类型来保存这类数据，你就只能利用上它整个取值范围的一半，而给那个数据列加上 UNSIGNED 属性之后，你的“活动范围”将立刻增加一倍。比如说，如果你想用某个数据列来保存序列编号，给它加上 UNSIGNED 属性将使可用编号的数量增加一倍。

你也可以给 DECIMAL 或浮点数据列加上 UNSIGNED 属性，但它的效果与整数数据列的情况稍有不同：浮点值的取值范围不会朝正数方向平移；原取值范围的正数部分将保持不变，而原取值范围的负数部分将全部变成 0（从效果上看，就像是把整个取值范围“砍掉”了一半儿）。

所有支持 UNSIGNED 属性的数值类型都可以用 SIGNED 属性来“修饰”。不过，因为那些数值类型在默认的情况下都是区分正负的，所以给它们加上 SIGNED 属性并没有什么实际效果。SIGNED 的作用

只是在数据列定义里明确地表明“这个数据列允许使用负数值”而已。

ZEROFILL 属性适用于除 BIT 类型以外的所有数值类型。如果设定了这一属性，这个数据列里的数值就会用一些前导的 0 来“加长”到这个数据列的显示宽度。如果想让数据列里的值全都整齐地按你设定的显示宽度被显示出来，就需要使用 ZEROFILL 属性。准确地讲，显示宽度其实只是“最小显示宽度”，因为那些长度大于显示宽度的数值将完整地显示出来而不会被截短。这可以从下面的示例里得到证明：


```
mysql> DROP TABLE IF EXISTS mytbl;
mysql> CREATE TABLE mytbl (my_zerofill INT(5) ZEROFILL);
mysql> INSERT INTO mytbl VALUES(1),(100),(10000),(1000000);
mysql> SELECT my_zerofill FROM mytbl;
+-------------+
| my_zerofill |
+-------------+
|       00001 |
|       00100 |
|       10000 |
|     1000000 |
+-------------+
```

请注意最末尾的那个值，它的长度大于数据列的显示宽度，它却被完整地显示了出来。

如果给某个数据列加上了 ZEROFILL 属性，它将自动地转换成一个 UNSIGNED 数据列。

另外一个属性，AUTO_INCREMENT，可用在整数或浮点数据类型上。这个属性的作用是生成一组独一无二的标识符或序列号码。当你把 NULL 值插入一个 AUTO_INCREMENT 数据列时，MySQL 会自动生成下一个序列编号并把它放到这个数据列里去。如果你没有做出另外的设定，AUTO_INCREMENT 数据列的编号值将从 1 开始，每新增一个数据行，这个编号值就会加 1。如果你在数据表里删除了一些数据行，这些编号值也会受到影响。换句话说，编号值是可以被再次使用的，但这是否真的会发生还要取决于具体的存储引擎。

每个数据表最多只能有一个 AUTO_INCREMENT 数据列，这个数据列还应该具备 NOT NULL 属性和索引。一般来说，最好是把 AUTO_INCREMENT 数据列声明为一个 PRIMARY KEY 或一个 UNIQUE 索引。此外，因为序列编号不可能是负数，所以你通常还应该给这个数据列加上 UNSIGNED 属性。比如说，下面这几条语句都可以用来声明一个 AUTO_INCREMENT 数据列：

```
CREATE TABLE ai (i INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY);
CREATE TABLE ai (i INT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE);
CREATE TABLE ai (i INT UNSIGNED NOT NULL AUTO_INCREMENT, PRIMARY KEY (i));
CREATE TABLE ai (i INT UNSIGNED NOT NULL AUTO_INCREMENT, UNIQUE (i));
```

在前两条语句里，索引信息作为数据列定义的一部分被给出；在后两条语句里，索引是在 CREATE TABLE 语句的一个子句里给出的。如果索引只包括 AUTO_INCREMENT 数据列，用或不用一个子句来声明它都可以。如果你想创建一个涉及多个数据列的索引，AUTO_INCREMENT 数据列只是其中之一，就必须使用一个子句。（这方面的例子在 3.4.2 节中的第 1 小节有好几个。）

可以把 AUTO_INCREMENT 数据列明确地声明为 NOT NULL。事实上，即使你省略了 NOT NULL，MySQL 也会自动地加上它。

我们将在 3.4 节进一步讨论 AUTO_INCREMENT 数据列的行为特点。

以上介绍的各种属性都是数值类数据列特有的。在数据列的声明定义里，在给出这些属性之后，

你还可以继续设定通用的 NULL 或 NOT NULL 属性，否则，其默认行为将是允许使用 NULL 值。

你还可以利用 DEFAULT 属性为数据列设定默认值。下面这个数据表包含有 3 个 INT 数据列，它们的默认值分别是-1、1 和 NULL：

```
CREATE TABLE t
(
  i1 INT DEFAULT -1,
  i2 INT DEFAULT 1,
  i3 INT DEFAULT NULL
);
```

如果数据列定义不包含任何 DEFAULT 子句，MySQL 将根据 3.2.3 节里描述的规则为它挑选一个默认值。

5. 挑选数值数据类型

在为数值数据列挑选数据类型时，你需要考虑数据的取值范围并挑选一个能够覆盖该范围的最小类型来使用。选择较大的类型会浪费存储空间，毫无必要地使数据表变得很大，会降低数据的处理效率。先说整数值，如果数据的取值范围很小，比如人的年龄或兄弟姐妹的人数，TINYINT 类型就是最佳选择。MEDIUMINT 类型可以用来表示好几百万个值，这对很多应用项目来说都已经足够了，但它在存储空间方面有额外开销。BIGINT 类型的取值范围最大，但它占用的存储空间将是最小的整数类型（INT）的两倍，所以我们应该只在确有必要时才使用它。再看浮点值，DOUBLE 类型所占用的存储空间是 FLOAT 类型的两倍。因此，如果你不需要非常高的精确度或者数据的取值范围不是非常大的话，用 FLOAT 类型代替 DOUBLE 类型可以节约一半的存储开销。

每个数值数据列的取值范围是由它的类型决定的。如果你试图把一个超出某数据列取值范围的值插入该数据列，结果将取决于是否已启用严格 SQL 模式。如果已启用，超出取值范围的值将导致一个错误；如果未启用严格模式，MySQL 将截短这个值，MySQL 会先把这个值替换为该数据列取值范围的上限值或下限值后再进行插入，同时生成一条警告消息。

数据值的截短处理是根据数据类型的取值范围而不是它的显示宽度进行的。比如说，一个声明为 SMALLINT(3)类型的数据列的显示宽度只有 3 个字符，但它的取值范围却是从−32 768 到 32 767。如果你想把值 12345 插入到这个数据列里，那么，因为 12345 虽然要比这个数据列的显示宽度更长，但仍落在该数据列的取值范围内，所以插入操作不会出现截短处理，这个值仍将以 12345 的形式被插入和检索。可值 99999 就不同了，它超出了这个数据列的取值范围，所以它在插入时将被截短为 32767，以后的检索操作所查到的结果也将是 32767。

对于定点或浮点数据列，如果将被存储的值比该数据列所能容纳的小数位数多，MySQL 将根据这个数据列的类型声明对小数点后面的数字进行取舍。比如说，如果你把 1.23456 插入到一个 FLOAT(8, 1)数据列里，其结果将是 1.2；如果你把这个值插入到一个 FLOAT(8, 4)数据列里，其结果将是 1.2346。也就是说，你必须根据自己对精确度的要求来选定一个小数点后面的位数。如果你的数据需要精确到千分之一，就不能把小数点后面的位数声明为两位。

3.2.5　字符串数据类型

MySQL 有好几种数据类型来存储字符串值。字符串通常用来存储和处理下面这样的值：

```
'N. Bertram, et al.'
'Pencils (no. 2 lead)'
```

```
'123 Elm St.'
'Monograph Series IX'
```

字符串可以用来表示任何一种值，从这个意义上讲，它可以说是最常用的类型之一。比如说，你可以用二进制字符串类型来存储二进制数据，如图像、声音、gzip 压缩工具的输出结果等。

表 3-10 列出了 MySQL 提供的用来声明各种字符串值数据列的类型以及它们的最大尺寸和存储空间要求。*M* 代表数据列值的最大长度（二进制字符串以字节为单位，非二进制字符串以字符为单位），*L* 代表某给定值以字节计算的实际长度，*w* 是相关字符集里最“宽”的字符所占用的字节数。BLOB 和 TEXT 类型各有几种变体，它们之间的主要区别在于它们所能容纳的字符串值的最大尺寸。

3

表3-10　字符串数据类型

类型定义	最大长度	存储空间占用量
BINARY[(*M*)]	*M*个字节	*M*个字节
VARBINARY(*M*)	*M*个字节	*L*+1 或 *L*+2 个字节
CHAR[(*M*)]	*M*个字符	*M* × *w*个字节
VARCHAR(*M*)	*M*个字符	*L*+1 或 *L*+2 个字节
TINYBLOB	2^8–1个字节	*L*+1个字节
BLOB	2^{16}–1个字节	*L*+2个字节
MEDIUMBLOB	2^{24}–1个字节	*L*+3个字节
LONGBLOB	2^{32}–1个字节	*L*+4个字节
TINYTEXT	2^8–1个字符	*L*+1个字节
TEXT	2^{16}–1个字符	*L*+2个字节
MEDIUMTEXT	2^{24}–1个字符	*L*+3个字节
LONGTEXT	2^{32}–1个字符	*L*+4个字节
ENUM（'*value1*', '*value2*', ……）	65535个成员	1或2个字节
SET（'*value1*', '*value2*', ……）	64个成员	1、2、3、4或8个字节

有些类型用来保存二进制字符串（字节串），其他类型用来保存非二进制字符串。因此，表 3-10 里列出的最大尺寸对二进制字符串类型来说是字节数，对非二进制字符串来说是字符数。比如说，BINARY(20)容纳 20 个字节，而 CHAR(20)容纳 20 个字符（由多字节字符构成的字符串的长度肯定会超过 20 个字节）。二进制字符串和非二进制字符串在字节和字符方面的语义区别请参见 3.1.2 节。每一种二进制字符串类型都有一种对应的非二进制字符串类型，如表 3-11 所示。

表3-11　二进制字符串和非二进制字符串的对应关系

二进制字符串类型	非二进制字符串类型
BINARY	CHAR
VARBINARY	VARCHAR
BLOB	TEXT

包括 ENUM 和 SET 类型在内的每一种非二进制字符串类型都可以被指定一种字符集和排序方式。不同的数据列可以被指定不同的字符集。关于如何指定字符集的讨论参见 3.2.5 节的第 5 小节。

BINARY 和 CHAR 是固定长度的字符串类型。对于这两种类型的数据列，MySQL 将为每个值分配同样数量的存储空间，如果某个值比数据列的长度短，则在其尾部添加一些零值字节（0x00，BINARY

类型）或空格（CHAR 类型）以补足之。因为 CHAR(*M*)数据列必须能够容纳由相关字符集里最宽的字符构成的字符串，所以该数据列需要占用 $M \times w$ 个字节，w 是字符集里最宽的字符所占用的字节数。比如说，ujis 字符集里的每个字符需要占用 1 到 3 个字节，所以 CHAR(20)必须分配 60 个字节以应对那 20 个字符全都需要占用 3 个字节的情况。

其他的字符串类型都是可变长度的，每个值占用的存储空间会随着数据行的不同而不同，并取决于实际存放到数据列里的值的长度。对于可变长度的类型，这个长度在表 3-10 里用 L 来代表。在 L 的基础上多出来的额外字节是用来保存有关数据的长度所必须的字节数。在对长度可变的数据进行处理时，MySQL 要把数据内容和数据长度都保存起来。这些额外的字节将被当做无符号整数来对待。可变类型的最大长度、对应于该类型的额外字节数以及占用同样字节个数的无符号整数的取值范围，这三者之间存在着一定的规律。比如说，MEDIUMBLOB 值的最大长度是 $2^{24}-1$ 个字节，这个长度值需要 3 个字节来存放，而占用 3 个字节的整数类型 MEDIUMINT 的最大无符号值就等于 $2^{24}-1$。它们之间的这种规律不是巧合。

对于 VARBINARY 和 VARCHAR 类型，如果数据列值以字节计算的最大长度小于 256，长度前缀将占用 1 个字节，否则将占用 2 个字节。

除 ENUM 和 SET 类型以外，所有字符串类型的值都存储为一串连续的字节，MySQL 将根据该类型容纳的是二进制字符串还是非二进制字符串把那些字节序列解释为一系列字节或一系列字符。如果数据值的长度超出了它们的表示范围，MySQL 将自动对有关数据进行截短处理。（在严格模式下，如果被截去的字符不是空格，MySQL 将报告一个错误。）不过，因为字符串类型有这么多种，取值范围从小到大一应俱全——最大的类型能够容纳将近 4GB 的数据，所以从理论上讲，你完全能够找到一种足够长的字符串类型来满足存储要求，而不需要 MySQL 对它们进行截短处理。（在实际工作中，字符串数据列的最大有效长度取决于 MySQL 所使用的“客户/服务器”通信协议所支持的数据包最大长度，它在默认的情况下是 1MB。）

ENUM 和 SET 类型的数据列定义里都有一个合法字符串值的列表，但 ENUM 和 SET 值在内部都被保存为数值，具体细节请参阅“ENUM 和 SET 数据类型”小节。在没有启用严格模式的情况下，试图把一个没有在列表里出现过的值放到 ENUM 或 SET 数据列里会导致该值被转换为一个空字符串（''）；在严格模式下，MySQL 将报告一个错误。

1. CHAR和VARCHAR数据类型

CHAR 和 VARCHAR 字符串类型用来保存非二进制字符串，因而与一种字符集和排序方式相关联。

CHAR 和 VARCHAR 数据类型的主要区别在于它们的长度是固定的还是可变的，以及它们如何对待尾缀的空格。

- CHAR 是一种固定长度的类型，而 VARCHAR 是一种长度可变的类型。
- 从 CHAR 数据列检索出来的值的尾缀空格将被去掉。给定一个 CHAR(*M*)数据列，如果某个值的长度小于 *M* 个字符，MySQL 在把它存入该数据列时将用空格把它补足到 *M* 个字符长，但那些追加的空格在检索时将被去掉。从 MySQL 5.1.20 版开始，你可以通过启用 PAD_CHAR_TO_FULL_LENGTH SQL 模式让 MySQL 保留从 CHAR 数据列检索出来的值的尾缀空格。
- 对于 VARCHAR(*M*)数据列，尾缀空格在存储和检索时都会被保留。

在定义 CHAR 数据列时，你可以把它的最大长度 M 定义为 1 到 255 之间的一个整数。CHAR 类型中的 M 是可选的，如果省略，它的默认值是 1。请注意，CHAR(0)是合法的，如果你允许它为 NULL 值，你还可以用它来表示 on/off 开关值。这种数据列只有两种可取值：NULL 值或空字符串。在数据表里，

CHAR(0)数据列只占用非常少的空间——只占用一位。

对 VARCHAR(*M*)数据列而言，*M* 在语义上的取值范围是 1 到 65 535，但它实际能够容纳的最大字符个数肯定小于 65 535——这是因为 MySQL 数据表里的每个数据行的最大长度只有 65 535 个字节。还要考虑以下因素。

- 一个长 VARCHAR 数据列需要 2 个字节来存放字符串值的长度，这两个字节计算在数据行总长度之内。
- 如果使用了多字节字符，数据行最大长度所能容纳的字符个数将减少。
- 数据表里往往还有其他的数据列，而那些数据列将挤占 VARCHAR 数据列的“生存”空间。

当你需要在 CHAR 和 VARCHAR 类型中作出选择时，请记住下面两个原则。

- 如果你的数据都是 M 个字符长，一个 VARCHAR(*M*)数据列将比一个 CHAR(*M*)数据列多占用一些存储空间，因为数据列里的每一个值还要多用一个或两个字节来保存其长度。反之，如果你的数据长短不一，选用 VARCHAR 类型就有节省存储空间的好处。CHAR(*M*)数据列总是占用 *M* 个字符的空间，即使它是空白字符串或 NULL 值。
- 如果数据在长度方面差别不大，且已经选定要使用 MyISAM 数据表，那么选用 CHAR 类型往往要比选用 VARCHAR 类型的效果好一些。这是因为 MyISAM 存储引擎对固定长度的数据行的处理效率要比对长度可变的数据行的处理效率高。请参阅 5.3 节。

说明 MySQL 5.0.3 及以前的版本对 VARCHAR 的处理和现在不一样。

- VARCHAR 类型的最大长度是 255。
- VARCHAR 值的尾缀空格在存储时会被去掉。

---

2. BINARY和VARBINARY数据类型

BINARY 和 VARBINARY 类型类似于 CHAR 和 VARCHAR，但有以下区别。

- CHAR 和 VARCHAR 是非二进制类型，MySQL 将根据相关的字符集和排序方式把 CHAR 和 VARCHAR 数据列里的值解释为一系列字符。比较操作的依据是各字符的先后顺序。
- BINARY 和 VARBINARY 是二进制类型，BINARY 和 VARBINARY 数据列里的值是一串字节，不涉及任何字符集和排序方式。比较操作的依据是各字节的数值大小。

对 BINARY 值的尾缀空格进行处理的规则如下所示。

- 在 MySQL 5.0.15 及更高的版本里，较短的值用 0x00 字节补足。在检索时不去掉任何东西。
- 在 MySQL 5.0.15 以前的版本里，较短的值用空格补足。在检索时去掉尾缀的空格。

对于 VARBINARY 类型，在存储时不补足，在检索时不截短。

3. BLOB和TEXT数据类型

BLOB 是英文 binary large object（二进制大对象）的字头缩写，其基本含义是指一个能够用来盛放任何东西的容器，而且是你想让它有多大，它就有多大。MySQL 里的 BLOB 类型其实是一个由 TINYBLOB、BLOB、MEDIUMBLOB 和 LONGBLOB 等类型组成的大家庭。除各自所能容纳的最大信息量不同以外，它们在其他方面完全相同（参见表 3-10）。BLOB 数据列存储的是二进制字符串，如果你想保存的信息有可能急剧膨胀到非常大的地步，或者各数据行的长短差异很大，就很适合用 BLOB 数据列来存放。压缩数据、加密数据、图像和声音都是很好的例子。

MySQL 还有一个由 TINYTEXT、TEXT、MEDIUMTEXT 和 LONGTEXT 等类型组成的 TEXT 家族，它们

和相应的 BLOB 类型有很多相似之处，但 TEXT 类型存储的是非二进制字符串而不是二进制字符串。换句话说，它们存储的是字符而不是字节，并且与某种字符集和排序方式相关联。二进制字符串和非二进制字符串的不同之处（请参见 3.1.2 节）就是因此而导致的。比如说，在比较操作中，BLOB 值以字节为单位被比较，而 TEXT 值则以字符为单位根据数据列的顺序被比较。

能否在 BLOB 和 TEXT 数据列上建立索引要取决于你具体使用的存储引擎，这里的原则有以下几点。

- MyISAM 和 InnoDB 存储引擎支持在 BLOB 和 TEXT 数据列上建立索引，但你必须为索引设定一个前缀长度。这是为了避免因用来建立索引项的原始数据过于庞大而抵消掉索引带来的好处。不过，如果你打算在 TEXT 数据列上建立一个 FULLTEXT 索引，就用不着为它设定前缀长度，因为基于 FULLTEXT 索引的检索操作将以带索引数据列里的完整内容为依据，即使你设定了前缀长度，也会被忽略。
- MEMORY 数据表不支持 BLOB 和 TEXT 索引，因为 MEMORY 存储引擎根本不支持 BLOB 和 TEXT 数据列。

BLOB 或 TEXT 数据列往往会有一些特殊的要求，如下所示。

- BLOB 和 TEXT 数据列里的值在长度方面往往差异巨大，所以 BLOB 和 TEXT 数据列的删除和修改操作很容易在数据表里产生大量碎片。如果使用了一个 MyISAM 数据表来保存 BLOB 或 TEXT 值，应该定期运行 OPTIMIZE TABLE 命令以减少碎片和改善系统性能，有关细节请参阅第 5 章。
- max_sort_length 系统变量的设置情况会影响 BLOB 和 TEXT 值的比较和排序操作：每个 BLOB 或 TEXT 值只有前 max_sort_length 个字节参加比较或排序。（如果某个 TEXT 数据列使用的是一个多字节字符集，这意味着参加比较的字符数肯定少于 max_sort_length。）如有可能因为 max_sort_length 的默认值 1024 不够大而导致问题，就应该在执行比较操作前加大这个值。
- 如果你正使用非常巨大的数值，就可能需要配置 MySQL 服务器以加大 max_allowed_packet 参数的值，有关细节请参阅 12.6.2 节。对于那些可能会用到非常巨大的数值的客户程序，你也许还需要加大它们的数据包大小，mysql 和 mysqldump 客户程序都支持直接使用一个启动选项来设置这个值。

4. ENUM和SET数据类型

ENUM 和 SET 是比较特殊的字符串数据类型，它们的取值只能是预先定义好的字符串。这两种类型的主要区别是：ENUM 数据列里必须包含且只包含一个来自它值列表的成员，而 SET 数据列里则允许包含任意多个来自值列表的成员（可以为空，也可以是全体成员）。换句话说，ENUM 类型的值不允许同时出现，而 SET 类型的值允许同时出现。

ENUM 数据类型定义的是枚举集合。ENUM 数据列里的值是且只能是你在创建数据表时定义的一个合法值。ENUM 类型的值列表最多允许有 65 535 个成员。ENUM 类型通常用来表示分组情况。比如说，如果你把一个数据列的值定义为 ENUM('N', 'Y')，那它就只能是'N'或'Y'。你还可以用 ENUM 类型来表示某种产品的尺寸或颜色、某调查问卷中的多重选择题的答案（每次只允许选择一个），如下所示：

```
employees ENUM('less than 100','100-500','501-1500','more than 1500')
color ENUM('red','green','blue','black')
size ENUM('S','M','L','XL','XXL')
```

```
vote ENUM('Yes','No','Undecided')
```

再举一个例子。很多网站都会在它们的 Web 网页上用一组单选按钮向访问者提供一些选项，这些选项完全能够用 MySQL 数据库里的 ENUM 数据列表示出来。如果你打算在网上提供比萨饼订购服务，就可以用一个 ENUM 数据列来表示顾客订购的比萨饼的口感和大小，如下所示：

```
crust ENUM('thin','regular','pan style','deep dish')
size ENUM('small','medium','large')
```

如果你打算用 ENUM 类型来表示统计数字的分布情况，就必须在创建其枚举集合时确定好分组标准，不要让统计范围出现重叠或者空缺。比如说，如果你想把实验中的白血球计数值的分布情况记录下来，可以像下面这样来分组：


```
wbc ENUM('0-100','101-300','>300')
```

实验的结果是一个具体的白血球计数值，这个计数值落在哪个分组范围内，你就把代表该范围的枚举值记到 wbc 数据列里。但要注意，这个过程是不可逆的，你无法把一个表示分组情况的 ENUM 数据列转换为一个保存准确数值的整数列。如果真的需要记录准确的白血球计数值，就应该使用一个整数列，然后在检索那些整数时用 CASE 语句对它们分组。比如说，如果把 wbc 定义为一个整数列，你可以像下面这样进行选取和分组：

```
SELECT CASE WHEN wbc <= 100 THEN '0-100'
            WHEN wbc <= 300 THEN '101-300'
            ELSE '>300' END AS 'wbc category'
FROM ...
```

SET 类型与 ENUM 类型的相似之处是：在创建 SET 数据列时，同样需要为它定义一个合法取值列表。SET 类型与 ENUM 类型的不同之处是：SET 数据列允许多个合法取值同时出现，而 ENUM 数据列只允许一个合法取值出现。SET 类型的合法取值列表最多允许有 64 个成员。如果数据的个数有限但它们有可能同时出现，就应该考虑使用一个 SET 数据列而不是一个 ENUM 数据列。比如说，你可以用一个 SET 类型来表示汽车的选装设备：

```
SET('luggage rack','cruise control','air conditioning','sun roof')
```

而这个 SET 数据列里的数据值则会根据顾客的具体要求产生多种组合，如下所示：

```
'cruise control,sun roof'
'luggage rack,air conditioning'
'luggage rack,cruise control,air conditioning'
'air conditioning'
''
```

最末尾的那个值是一个空字符串，它表示顾客没有订购任何选装设备。空字符串也是一个合法的 SET 值。

在定义 SET 数据列时，要把所有的可取值以逗号分隔写成一个字符串列表，列表中的各个元素就是该集合的成员。SET 数据列的值必须是一个字符串，如果它由集合里的多个成员构成，必须用逗号把这个字符串里的各个成员分隔开。这意味着我们不能用一个包含逗号的字符串作为一个 SET 成员。

可以用 SET 类型来表示各种信息，如病人的症状、来自 Web 页面的选择结果等。就拿病人的症状来说吧，很多疾病都有一个标准的症状清单，医生们会按照这个清单来询问病人是否有那些症状，病人可能只有一种症状，也可能同时有好几种（甚至全部）症状：

```
SET('dizziness','shortness of breath','cough')
```

再来看看你的网上比萨饼店，你在 Web 页面上安排了一组多选按钮供顾客们自由选择比萨饼的馅料，顾客可以只选一种馅，也可以选择好几种馅：

```
SET('pepperoni','sausage','mushrooms','onions','ripe olives')
```

在为 ENUM 或 SET 数据列定义合法取值列表时，必须考虑到以下几个因素。

- 这个列表将决定数据列可能的合法取值，这一点我们刚才已经讨论过了。
- 如果 ENUM 或 SET 数据列有一个不区分大小写的排序方式，你在插入合法值时就用不着区分字母的大小写；但当你检索 ENUM 或 SET 数据列里的数据时，它们将按照数据列定义的合法取值列表里的字母大小写情况来显示。比如说，你定义了一个 ENUM('N', 'Y')数据列，那么，在插入操作中，你完全可以使用'n'和'y'，但在检索操作中，它们却显示为'N'和'Y'。如果这个数据列有一个区分大小写的排序方式或二进制排序方式，就必须严格按照你在定义该数据列时使用的大小写字母来插入数据；否则，MySQL 将认为它们是非法的。反过来说，只要你使用的排序方式区分大小写，你就可以把多个只在大小写方面有区别的成员定义在同一个 ENUM 或 SET 里。
- 在声明 ENUM 数据列时，其中的合法取值出现的次序将是这个数据列上的排序操作所使用的顺序。SET 数据列上的排序操作也遵守同样的规则，但因为 SET 数据列允许多个合法取值同时出现，所以情况可能要复杂得多。
- 如果 SET 数据列里同时出现了多个合法取值，那么，在检索结果里，它们将按在数据列的出现次序被显示出来。

在创建 ENUM 或 SET 数据列时，你需要以字符串的形式列出枚举和集合成员，因此，我们把 ENUM 和 SET 划分为字符串类的数据列类型。但是，这些成员在 MySQL 的内部是以数值形式存储的，你对它们的操作也将按数值操作来对待。这意味着 ENUM 和 SET 类型要比其他字符串类型有着更好的处理性能，因为它们可以用数值操作而不是字符串操作来处理。这同时也意味着 ENUM 和 SET 值既能够用在字符串上下文里，也能够用在数值上下文里。最后要提醒大家一句：如果你在字符串上下文里使用 ENUM 和 SET 数据列，但希望它们的行为像数值那样（或情况刚好相反），它们可能会引起混乱。

MySQL 将按照 ENUM 数据列的合法取值在声明中的先后顺序对它们编号，从 1 开始。（编号 0 是 MySQL 保留的出错代码，这个出错代码的字符串形式是一个空字符串。）ENUM 数据列占用的存储空间取决于枚举值的个数。用一个字节可以表示 256 个值，用两个字节就能表示 65 536 个值。（这恰好分别是占用一个字节的整数类型 TINYINT UNSIGNED 和占用两个字节的整数类型 SMALLINT UNSIGNED 的取值范围。）因此，一个 ENUM 数据列的合法取值列表最多允许有 65 536 个成员（包括编号为 0 的出错代码在内），它占用的存储空间是一个字节还是两个字节则要视它的合法取值是否多于 256 个而定。因为 MySQL 需要为出错代码预留一个位置且每一个 ENUM 数据列的合法取值列表都隐含这个出错成员，所以你能在 ENUM 声明里给出的合法取值的最大个数就不是 65 536 而是 65 535。如果你试图把一个非法取值放入一个 ENUM 数据列，MySQL 就会把它替换为编号为 0 的出错成员（在严格模式下，MySQL 将抛出一个错误而不是进行这样的替换）。

下面这个例子演示了以字符串方式和数值方式来检索 ENUM 数据值的操作情况（合法取值有数值编号而 NULL 值没有数值编号）：

```
mysql> CREATE TABLE e_table (e ENUM('jane','fred','will','marcia'));
mysql> INSERT INTO e_table
    -> VALUES('jane'),('fred'),('will'),('marcia'),(NULL);
```

```
mysql> SELECT e, e+0, e+1, e*3 FROM e_table;
+--------+------+------+------+
| e      | e+0  | e+1  | e*3  |
+--------+------+------+------+
| jane   |    1 |    2 |    3 |
| fred   |    2 |    3 |    6 |
| will   |    3 |    4 |    9 |
| marcia |    4 |    5 |   12 |
| NULL   | NULL | NULL | NULL |
+--------+------+------+------+
```

ENUM 值的比较操作可以按名字或数值编号来进行：

```
mysql> SELECT e FROM e_table WHERE e='will';
+------+
| e    |
+------+
| will |
+------+
mysql> SELECT e FROM e_table WHERE e=3;
+------+
| e    |
+------+
| will |
+------+
```

MySQL 允许把空字符串定义为 ENUM 数据列的成员。作为一个成员的空字符串将分配到一个非零的数值编号，就像 ENUM 数据列的其他成员一样。但是，把空字符串当做一个成员往往会引起混乱，因为空字符串还充当编号为 0 的出错成员。在下面的例子里，我们试图往 ENUM 数据列里插入一个非法取值'x'，这将导致 MySQL 插入一个出错成员。如果按字符串方式操作，我们将无法分辨空字符串究竟表示一个合法成员还是表示出现了一个错误；但如果按数值方式进行操作，我们就能看得很清楚了。

```
mysql> CREATE TABLE t (e ENUM('a','','b'));
mysql> INSERT INTO t VALUES('a'),(''),('b'),('x');
mysql> SELECT e, e+0 FROM t;
+------+------+
| e    | e+0  |
+------+------+
| a    |    1 |
|      |    2 |
| b    |    3 |
|      |    0 |
+------+------+
```

在严格模式下，插入非法取值'x'将导致一个错误而不会插入任何值。

SET 数据列的数值表示方法与 ENUM 数据列稍有不同。SET 成员并不是按顺序编号的。每个 SET 成员对应着 SET 值里的一个位。第一个成员对应着第 0 位，第二个成员对应着第 1 位，依次类推。换句话说，SET 成员的对应数值都是 2 的幂，空字符串对应着数值为 0 的 SET 成员。

SET 值被存储为位值。每个字节对应 8 个 SET 成员，所以一个 SET 数据列占用的存储空间取决于它的成员的个数，最多只能有 64 个。因此，如果一个 SET 数据列有 1 到 8、9 到 16、17 到 24、25 到 32、33 到 64 个合法值，这个 SET 数据列就将占用 1、2、3、4 或 8 个字节。

正是因为一个 SET 值由一组位表示，所以一个 SET 值可以由多个 SET 成员组成。同时，因为值中可以出现任意组合的位，所以值应是 SET 定义中与这些位对应的字符串的某种组合。

下面这个示例演示了 SET 数据列的字符串形式与它的数值形式之间的关系。我们把数值形式的 SET 值分别显示为十进制数和二进制数：

```
mysql> CREATE TABLE s_table (s SET('table','lamp','chair','stool'));
mysql> INSERT INTO s_table
    -> VALUES('table'),('lamp'),('chair'),('stool'),(''),(NULL);
mysql> SELECT s, s+0, BIN(s+0) FROM s_table;
+-------+------+----------+
| s     | s+0  | BIN(s+0) |
+-------+------+----------+
| table |    1 | 1        |
| lamp  |    2 | 10       |
| chair |    4 | 100      |
| stool |    8 | 1000     |
|       |    0 | 0        |
| NULL  | NULL | NULL     |
+-------+------+----------+
```

当你把值'lamp,stool'插入数据列 s 时，MySQL 将在其内部把它保存为 10（二进制数 1010），因为'lamp'的对应数值是 2（第 1 位），'stool'的对应数值是 8（第 3 位）。

在对 SET 数据列赋值时，你可以按任意顺序来安排各个子串，而不必照搬它们在值列表里的先后顺序。不过，在检索操作中，各子串将按它们在声明里的先后顺序显示出来。此外，在 SET 数据列的赋值操作中，如果被插入的值里包含有不是该数据列合法值的子字符串，MySQL 就会把它们剔除掉，只把剩余的子串赋给那个 SET 数据列。在检索那个值时，非法子串将不会出现。

如果你把'chair, couch, table'赋值给 s_table 数据表里的数据列 s，将会发生两件事。

- 'couch'将被剔除，因为它不是该数据列的一个合法值。之所以会如此，是因为在赋值操作中，SET 值中的位是由 MySQL 根据它们与各值的对应关系而置位或者清零的。因为不存在与'couch'相对应的位，所以它被剔除掉了。
- 当你在今后检索到这个值时，它将显示为'table, chair'。在检索操作中，MySQL 按顺序扫描各个位，根据数值构造字符串值，按照定义数据列时的顺序对子字符串自动重排列。重排序还意味着：即使你把一个值多次赋值给一个 SET 数据列，它也只会在检索结果里出现一次。也就是说，即使你把'lamp, lamp, lamp'赋值给 SET 数据列，你的检索结果里也只会有一个'lamp'。

在严格模式下，使用非法 SET 成员将导致错误，并且那个值将不会被保存。在刚才的例子里，插入一个包含'couch'的值将导致一个错误，整个赋值操作将失败。

在 MySQL 里，SET 数据列值的重排序会导致这样一种结果：如果你想用字符串来检索某个值，就必须按适当的顺序列出各子字符串。仍以上面那个 SET 数据列为例，你可以在插入操作中使用'chair, table'，但如果在检索操作中也使用'chair, table'，就会找不到这条记录——你必须用'table, chair'查找。

ENUM 和 SET 数据列上的排序和索引操作都是以数据列取值的内部值（即数值形式的值）为依据的，与人们惯见的顺序（如字母表顺序）可能会不一样。例如，下面这个例子看起来好 像不正确，因为输出结果没有以字母表顺序显示：

```
mysql> SELECT e FROM e_table ORDER BY e;
+--------+
| e      |
+--------+
| NULL   |
|        |
| jane   |
| fred   |
| will   |
| marcia |
+--------+
```

同时对 ENUM 值的字符串和数值形式进行检索可以让你看清楚到底发生了什么：

```
mysql> SELECT e, e+0 FROM e_table ORDER BY e;
+--------+------+
| e      | e+0  |
+--------+------+
| NULL   | NULL |
|        |    0 |
| jane   |    1 |
| fred   |    2 |
| will   |    3 |
| marcia |    4 |
+--------+------+
```

如果你的数据是一个有限集合，你就可以利用 ENUM 类型上的排序特点把它们按你希望的顺序进行排序和输出。具体做法是：先声明一个 ENUM 数据列，把有关数据按你希望的顺序依次写在 ENUM 数据列里。假设你有一个用来存放橄榄球队成员个人资料的数据表，你想按这些成员的场上位置（如按教练、助理教练、四分卫、跑锋、接球手、边线防守队员这样的顺序）对查询结果进行排序。于是，你定义了一个 ENUM 数据列，并把场上位置按你想要的顺序列在它的取值列表里。这样，该 ENUM 数据列上的排序操作就能按你设定的顺序自动生成相应的查询结果了。

如果你想让某个 ENUM 数据列上的排序操作按正常的字母表顺序输出，可以先用 CAST()函数把这个数据列里的值转换为一个非 ENUM 字符串，然后再排序，如下所示：

```
mysql> SELECT CAST(e AS CHAR) AS e_str FROM e_table ORDER BY e_str;
+--------+
| e_str  |
+--------+
| NULL   |
|        |
| fred   |
| jane   |
| marcia |
| will   |
+--------+
```

CAST()函数并没有真正改变 ENUM 数据列里的数据值，它在这条语句里的作用是把查询结果中的 ENUM 值转换为正常的字符串，从而改变它们的排序输出效果。

5. 字符串数据类型的属性

字符串数据类型特有的属性是 CHARACTER SET（或 CHARSET）和 COLLATE，它们分别用来指定一种字符集和一种排序方式。你可以为整个数据表指定一种默认的字符集和排序方式，也可以为各数据

列分别设置这两个属性，重写整个数据表的默认设置。（事实上，每个数据库也有默认的字符集和排序方式，就连服务器也是如此。这些默认设置将在你创建数据表时发挥作用，我们马上就会看到。）

CHARACTER SET 和 COLLATION 属性适用于 CHAR、VARCHAR、TEXT、ENUM 和 SET 数据类型，但不适用于二进制字符串数据类型（BINARY、VARBINARY 和 BLOB），因为那些类型的内容是一些字节串，而不是一些字符串。

在设定 CHARACTER SET 和 COLLATION 属性时，不管是在数据列、数据表还是在数据库级别上设置，都要遵守以下规则。

- 你打算使用的字符集必须是 MySQL 支持的。SHOW CHARACTER SET 语句可以查出当前可用的字符集都有哪些。
- 如果在定义里同时使用了 CHARACTER SET 和 COLLATION 属性，它们所代表的字符集和排序方式必须是兼容的。比如说，如果字符集是 latin2，你可以使用 latin2_croatian_ci 排序方式，但不能使用 latin1_bin 排序方式。SHOW COLLATION 语句可以把每一种字符集支持的排序方式列出来。
- 如果在定义某个数据列时只给出了 CHARACTER SET 属性，而没有给出 COLLATION 属性，该数据列将使用默认排序方式。
- 如果在定义某个数据列时只给出了 COLLATION 属性，而没有给出 CHARACTER SET 属性，MySQL 将根据排序方式名称的第一部分来确定使用哪一种字符集。

上述规则的实际效果可以从下面这条语句的执行结果看出来。它创建了一个数据表，使用了好几种字符集：

```
CREATE TABLE mytbl
(
  c1  CHAR(10),
  c2  CHAR(40) CHARACTER SET latin2,
  c3  CHAR(10) COLLATE latin1_german1_ci,
  c4  BINARY(40)
) CHARACTER SET utf8;
```

由这条语句创建的数据表将使用 utf8 作为默认字符集。因为没有给出数据表级的 COLLATION 选项，所以默认的将是 utf8 字符集的默认排序方式（即 utf8_genaral_ci）。c1 数据列的定义里没有包含任何它自己的 CHARACTER SET 或 COLLATION 属性，所以它将使用数据表的默认设置。数据表级的字符集和排序方式不适用于 c2、c3 和 c4 数据列，因为 c2 和 c3 有它们自己的字符集信息，c4 是一种二进制字符串类型，字符集属性不适用于它。c2 数据列的排序方式是 latin2_general_ci，这是 latin2 字符集的默认排序方式。c3 数据列的字符集是 latin1，这是根据该数据列的排序方式的名字 latin1_german_ci 确定的。

如果想查看某个现有数据表的字符集信息，可以使用 SHOW CREATE TABLE 语句：

```
mysql> SHOW CREATE TABLE mytbl\G
*************************** 1. row ***************************
       Table: mytbl
Create Table: CREATE TABLE `mytbl` (
  `c1` char(10) default NULL,
  `c2` char(40) character set latin2 default NULL,
  `c3` char(10) character set latin1 collate latin1_german1_ci default NULL,
  `c4` binary(40) default NULL
```

```
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

如果 SHOW CREATE TABLE 语句没有显示某个数据列的字符集，就表明它与数据表的默认字符集相同。如果 SHOW CREATE TABLE 语句没有显示某个数据列的排序方式，该数据列的排序方式将与该数据列的字符集的默认排序方式相同。

还可以通过给 SHOW COLUMNS 语句加上 FULL 关键字来显示排序方式信息（并推断出它们的字符集）：

```
mysql> SHOW FULL COLUMNS FROM mytbl;
+-------+------------+-------------------+------+-----+---------+...
| Field | Type       | Collation         | Null | Key | Default |...
+-------+------------+-------------------+------+-----+---------+...
| c1    | char(10)   | utf8_general_ci   | YES  |     | NULL    |...
| c2    | char(40)   | latin2_general_ci | YES  |     | NULL    |...
| c3    | char(10)   | latin1_german1_ci | YES  |     | NULL    |...
| c4    | binary(40) | NULL              | YES  |     | NULL    |...
+-------+------------+-------------------+------+-----+---------+...
```

刚才的讨论只提到了数据列和数据表，其实字符集可以关联到数据列、数据表、数据库和服务器等多个级别。在处理字符数据列的定义时，MySQL 将依次根据下述原则为它指定一个字符集。

(1) 如果数据列的定义里指定了一个字符集，就使用这个字符集。（包括只给出了 COLLATON 属性的情况，因为 MySQL 可以根据排序方式的名字确定应该使用哪一种字符集。）

(2) 否则，如果数据表的定义里有一个数据表级的字符集选项，就使用那个字符集。

(3) 否则，使用数据库的字符集作为数据表的默认字符集，该字符集还将成为数据列的默认字符集。如果以前没有为数据库明确地指定过一个字符集（比如说，用 MySQL 4.1 以前的版本创建的数据库就是这样），数据库的字符集将沿用服务器的字符集。

换句话说，MySQL 会逐步扩大它为字符数据列搜索其关联字符集的范围，直到它找到一个经过定义的字符集，并将其用作该字符数据列的字符集。数据库永远有一个默认的字符集，所以即使你在任何低级别上都没有指定一个字符集，这一搜索过程也肯定会在到达数据库级别之后划上句号。

使用 binary 作为字符集的名字有特殊含义。为一个非二进制字符串数据列指定 binary 字符集相当于把该数据列定义为相应的二进制字符串类型。在下面几组数据列定义里，每一组里的两条语句是等价的：

```
c1 CHAR(10) CHARACTER SET binary
c1 BINARY(10)

c2 VARCHAR(10) CHARACTER SET binary
c2 VARBINARY(10)

c3 TEXT CHARACTER SET binary
c3 BLOB
```

如果你在定义二进制字符串数据列时使用了 CHARACTER SET binary 短语，它将被忽略，因为它已经是二进制类型了。如果你在定义 ENUM 或 SET 数据列时使用了 CHARACTER SET binary 短语，它将像前面讨论的那样生效或不生效。

如果把 binary 字符集赋值给一个数据表选项，它将在每一个在其自身的定义里没有给出任何字符集信息的字符串数据列上发生作用。

MySQL 还提供了一些简写形式用于定义字符数据列：

- ASCII 属性是 CHARACTER SET latin1 的简写形式。
- UNICODE 属性是 CHARACTER SET ucs2 的简写形式。
- 如果在定义非二进制字符串、ENUM 或 SET 数据列时使用了 BINARY 属性，数据列将使用其字符集的二进制排序方式。比如说，假设某个数据表的默认字符集是 latin1，以下定义是等效的：

```
c1 CHAR(10) BINARY
c2 CHAR(10) CHARACTER SET latin1 BINARY
c3 CHAR(10) CHARACTER SET latin1 COLLATE latin1_bin
```

  如果为二进制字符串数据列给出了 BINARY 属性，它将被忽略，因为它已经是二进制类型了。

任何一种字符串类型都能使用通用的 NULL 或 NOT NULL 属性。如果没有指定，NULL 将是默认的设置。不过，把一个字符串数据列声明为 NOT NULL 并不意味着它不能把空字符串（''）作为它的一项数据。在 MySQL 里，空值并不等于没有值，因此，千万不要因为你把一个字符串数据列声明成 NOT NULL 而错误地认为它将包含非空值。如果你想让字符串值全都是非空值，就必须在你的应用程序里明确做出这样的限制。

除 BLOB 和 TEXT 类型外，其他的字符串数据类型都可以用 DEFAULT 子句设定一个默认值。如果某个数据列定义没有包含任何 DEFAULT 子句，MySQL 将根据 3.2.3 节里描述的规则为它指定一个。

6. 挑选字符串数据类型

在为一个字符串数据列挑选数据类型时，请认真考虑以下问题。

应该把值表示为字符数据还是二进制数据？如果答案是字符数据，非二进制字符串类型最合适；如果答案是二进制数据，就应该选用一种二进制字符串类型。

比较操作需要区分字母的大小写情况吗？如果是，则应该选用一种非二进制字符串类型，让存储在数据库里的字符与一种字符集和排序方式关联起来。

非二进制字符串值在比较和排序操作中的大小写区分情况取决于你为它们指定的排序方式。如果你不想区分字母的大小写情况，就应该选用一种不区分大小写的排序方式；否则，就应该选用一种二进制或区分大小写的排序方式。二进制排序方式在比较两个字符时使用的是它们的数值编码。区分大小写的排序方式在比较两个字符时使用的是该种排序方式特有的字符顺序，这种顺序不见得和字符编码的大小顺序相同。在这两种情况下，某给定字符的小写和大写版本在比较操作中将被认为是不同的。假设'mysql'、'MySQL'和'MYSQL'是 latin1 字符集里的字符串。如果使用一种不区分大小写的排序方式（如 latin1_swedish_ci），它们将被认为是相同的；如果使用二进制排序方式（latin1_bin）或一种区分大小写的排序方式（如 latin1_general_cs），它们将被认为是不同的。

如果你需要对数据列里的数据分别进行区分大小写的比较和不区分大小写的比较，请使用你最经常进行的比较所用的排序方式。等你需要进行其他类型的比较时，利用 COLLATION 操作符临时改变一下排序方式就行了。比如说，假设 mycol 是一个使用 latin1 字符集的 CHAR 数据列，你可以挑选 latin1_swedish_ci 作为它的默认排序方式以进行不区分大小写的比较。下面的比较操作是不区分大小写的：

```
mycol = 'ABC'
```

当你需要进行区分大小写的比较时，可以临时改用 latin1_general_cs 或 latin1_bin 排序方式。下面的比较操作是区分大小写的（让 COLLATION 操作符作用在左操作数上或右操作数上都可以，

这不会影响到比较操作的结果）：

```
mycol COLLATE latin1_general_cs = 'ABC'
mycol COLLATE latin1_bin = 'ABC'
mycol = 'ABC' COLLATE latin1_general_cs
mycol = 'ABC' COLLATE latin1_bin
```

你想尽量少占用存储空间吗？如果是，选用一种可变长度的类型，不要选用固定长度的类型。

数据列的可取值总是某几个合法值之一或它们的组合吗？如果是，ENUM 或 SET 类型往往是最好的选择。

如果字符串数据是一个有限的集合并且你想按照某种非字母表顺序对它们进行排序，ENUM 类型往往可以帮上你的大忙。ENUM 值的排序结果取决于数据列定义里枚举值的出现顺序，你可以让 ENUM 值按照你希望的任何顺序排序。

尾缀的空格（或零值字节）很重要吗？如果数据必须原样存入数据库、原样取出数据库，在存储和检索过程中不增加和消除任何尾缀的空格（对二进制字符串而言是零值字节 0x00），你应该为非二进制字符串选用一个 TEXT 或 VARCHAR 数据列、为二进制字符串选用一个 BLOB 或 VARBINARY 数据列。这一点在你打算把压缩数据、散列数据或加密数据之类的东西存入数据库时非常关键，因为编码方法可能会在数据的末尾生成一串空格（或零值字节）。表 3-12 对各种字符串数据类型在存储和检索数据时对尾缀空格（或零值字节）的处理行为进行了汇总。

从 MySQL 5.1.20 版开始，你可以通过启用 PAD_CHAR_TO_FULL_LENGTH SQL 模式的办法让 MySQL 保留从 CHAR 数据列检索出来的值的尾缀空格。对于 MySQL 5.0.15 版本之前的 BINARY 数据列，较短的值在存储时将用空格补足到数据列宽度，在检索时将去掉那些补足的空格。

表3-12 字符串数据类型对尾缀空格（或零值字节0x00）的处理

数据类型	存 储 时	检 索 时	结 果
CHAR	补足	去掉尾缀	检索出来的值没有尾缀
BINARY	补足	无任何动作	检索出来的值没有尾缀
VARCHAR，VARBINARY	无任何动作	无任何动作	尾缀没有任何变化
TEXT，BLOB	无任何动作	无任何动作	尾缀没有任何变化

3.2.6 日期/时间数据类型

MySQL 提供的日期/时间数据类型有以下几种：DATE、TIME、DATETIME、TIMESTAMP 和 YEAR。表 3-13 列出了这些类型的合法取值范围。表 3-14 列出了这些类型的存储空间要求。

表3-13 日期/时间数据类型

类型定义	取值范围
DATE	'1000-01-01' 到 '9999-12-31'
TIME	'-838:59:59' 到 '838:59:59'
DATETIME	'1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'
TIMESTAMP	'1970-01-01 00:00:01' 到 '2038-01-19 03:14:07'
YEAR [(M)]	YEAR (4)：1901到2155；YEAR (2)：1970到2069

表3-14　日期/时间数据类型的存储空间要求

类型定义	存储空间
DATE	3字节
TIME	3字节
DATETIME	8字节
TIMESTAMP	4字节
YEAR	1字节

在对日期/时间值进行修改时，如果你插入的是一个非法值，MySQL 就会把它替换为相应的零值。表 3-15 列出了各种日期/时间类型的零值，它们同时也是被声明为有 NOT NULL 属性的日期/时间数据列的默认值。如果启用了 SQL 模式，非法值将导致 MySQL 抛出一个错误并拒绝它们进入数据库，详见 3.3 节。

表3-15　日期/时间类型的零值

类型定义	零　　值
DATE	'0000-00-00'
TIME	'00:00:00'
DATETIME	'0000-00-00 00:00:00'
TIMESTAMP	'0000-00-00 00:00:00'
YEAR	0000

依照 ANSI SQL 和 ISO 8601 等标准，MySQL 总是把日期/时间值里的年份数字放在最前面。比如说，2008 年 12 月 3 日将被表示为'2008-12-03'。但 MySQL 也允许在输入日期值的时候偷一点儿懒。比如说，你可以只输入年份的后两位数字，它会把这个两位数转换为四个数字的年份值。再比如说，你可以在输入小于 10 的月份和日期值时省略前导的数字 0。但无论如何，你都必须把年份数字放在最前面，把日期数字放在最末尾。对于人们日常生活中的一些习惯性写法（如'12/3/99'或'3/12/99'），MySQL 的解释恐怕和你想象的不一样。此时，STR_TO_DATE()函数可以帮你把非 ISO 格式的字符串转换为 ISO 格式的日期/时间值。我们将在这里的第 5 小节对 MySQL 解释日期/时间值的规则做进一步的讨论。

对于日期加时间形式的组合值，可以在日期和时间之间使用字母“T”来代替空格作为分隔符（例如，'2008-12-3T12:00:00'）。

时间值和日期/时间组合值可以包括一个毫秒部分，它出现在时间值的最末尾，由一个小数点和最多 6 位数字构成。（例如，'12:30:15.5' 或 '2008-06-15 10:30:12.000045'。）不过，MySQL 目前对毫秒的支持还不是很完备：虽然有一部分日期/时间函数可以对毫秒进行识别和处理，但目前还不能把带毫秒部分的日期时间值存入数据表，毫秒部分将被删除。

在检索时，你可以利用 DATA_FORMAT()和 TIME_FORMAT()函数以各种各样的格式来显示日期和时间值。

1. DATE、TIME和DATATIME数据类型

DATE 和 TIME 类型分别用来保存日期值和时间值，DATATIME 类型用来保存日期和时间的组合值。这 3 种类型的格式分别是'*CCYY-MM-DD*'、'*hh:mm:ss*'和'*CCYY-MM-DD hh:mm:ss*'，其中的 *CC*、*YY*、*MM*、*DD*、*hh*、*mm*、*ss* 分别代表世纪、年、月、日、时、分、秒。

在 DATETIME 类型里，日期值和时间值两部分都必不可少。如果把一个 DATE 值赋值给一个 DATETIME 数据列，MySQL 将自动补足时间值'00:00:00'；如果把一个 DATETIME 值赋值给一个 DATE 或 TIME 数据列，MySQL 将去掉它里面的时间值：

```
mysql> CREATE TABLE t (dt DATETIME, d DATE, t TIME);
mysql> INSERT INTO t (dt,d,t) VALUES(NOW(), NOW(), NOW());
mysql> SELECT * FROM t;
+---------------------+------------+----------+
| dt                  | d          | t        |
+---------------------+------------+----------+
| 2007-09-14 10:26:26 | 2007-09-14 | 10:26:26 |
+---------------------+------------+----------+
```

在 MySQL 里，DATATIME 类型里的时间值与 TIME 值是有区别的。DATATIME 类型代表的是几点几分，它必须落在'00:00:00'到'23:59:59'的范围内。TIME 值代表的是一段逝去了的时间——这正是 TIME 数据列的取值可以大于'23:59:59'以及可以是负数的原因（请参见表 3-13）。

在往数据表里插入 TIME 值时，如果你输入的是一个不完整的“短”值，MySQL 对它的解释可能会出乎你的预料。比如说，如果你把'30'和'12:30'都插入到一个 TIME 数据列里，就会发现它们一个被解释为'00:00:30'，而另一个却被解释为'12:30:00'。如果你输入'12:30'时想表达的真正意思是“12 分 30 秒”，就应该把它完整地写成'00:12:30'的样子。

2. TIMESTAMP数据类型

TIMESTAMP 数据类型用来保存日期和时间的组合值。（单词“timastamp”容易让人误以为这种数据类型只与时间有关，但实际情况却并非如此。）TIMESTAMP 数据类型有一些特殊的属性，我们将在下面讨论。

TIMESTAMP 数据列的取值范围是从'1970-01-01 00:00:00'到'2038-01-19 03:14:07'。这个时间区间与 Unix 时间密切相关：Unix 系统把 1970 年的第一天当做“日期零点”或“纪元始点”，并用 4 个字节来保存从纪元起点开始以秒计算的时间。TIMESTAMP 类型把即将进入 1970 年的那一刻作为它取值范围的下限值，上限值对应着 4 个字节所能表示的 Unix 时间的极限。

TIMESTAMP 值以 UTC（Universal Coordinated Time，世界标准时间）格式存放。当你保存一个 TIMESTAMP 值的时候，服务器会把它从连接的时区转换为 UTC 时间值。当你以后检索这个值的时候，服务器又会把它从 UTC 时间值转换回连接的时区，让你看到的时间值和你存储的一样。不过，如果另一个客户使用另一个时区去连接服务器并检索这个值，它将看到按照它自己的时区转换而来的值。事实上，只要在你自己的连接里改变了时区设置，就可以看到这样的效果：

```
mysql> CREATE TABLE t (ts TIMESTAMP);
mysql> SET time_zone = '+00:00';   # set time zone to UTC
mysql> INSERT INTO t VALUES('2000-01-01 00:00:00');
mysql> SELECT ts FROM t;
+---------------------+
| ts                  |
+---------------------+
| 2000-01-01 00:00:00 |
+---------------------+
mysql> SET time_zone = '+03:00';   # advance time zone 3 hours
mysql> SELECT ts FROM t;
+---------------------+
| ts                  |
```

```
+---------------------+
| 2000-01-01 03:00:00 |
+---------------------+
```

在上面的例子里，时区设置是以一个相对于 UTC 标准时间的偏移值的形式给出的。如果服务器的时区数据表已经按照 12.9.1 节里的步骤设置好了，你还可以使用'Europe/Zurich'这样的时区名来改变时区设置。

TIMESTAMP 类型具备自动初始化和自动更新属性。可以让数据表里的任何一个 TIMESTAMP 数据列同时具备这两种属性或只具备其中之一。

- "自动初始化"的含义是：如果插入新数据行时没有在 INSERT 语句里给出 TIMESTAMP 数据列的值或是把它设置成了 NULL，该数据列将自动取值为当前的时间戳。
- "自动更新"的含义是：当你修改某个现有数据行的其他数据列时，TIMESTAMP 数据列将自动更新为当前的日期和时间。请注意，把数据列设置为它的当前值不算是"修改"；你必须把它修改为一个不同的值才能触发 TIMESTAMP 数据列的自动更新行为。

此外，对于任何一个 TIMESTAMP 数据列，如果把它设置为 NULL，它就会自动取值为当前的时间戳值。但如果给一个 TIMESTAMP 数据列定义了 NULL 属性，就可以把 NULL 值存入该数据列而抑制其自动更新行为。

每个数据表只允许一个 TIMESTAMP 数据列具备上述自动属性。你无法让一个 TIMESTAMP 数据列具备自动初始化属性，让另一个 TIMESTAMP 数据列具备自动更新属性。你也无法让同一个数据表里的多个 TIMESTAMP 数据列同时具备自动初始化或自动更新属性。

下面是 TIMESTAMP 数据列的定义语法，假设该数据列的名字是 ts：

```
ts TIMESTAMP [DEFAULT constant_value] [ON UPDATE CURRENT_TIMESTAMP]
```

如果需要同时给出 DEFAULT 和 ON UPDATE 属性，它们的先后顺序无关紧要。TIMESTAMP 数据列的默认值可以是 CURRENT_TIMESTAMP 、一个常数值（如 0）或一个'*CCYY-MM-DD hh:mm:ss*'格式的值。函数形式的 CURRENT_TIMESTAMP()和 NOW()是 CURRENT_TIMESTAMP 的同义词，它们可以在 TIMESTAMP 数据列的定义里互换使用。

为了让数据表里的第一个 TIMESTAMP 数据列同时具备两种自动属性或其中之一，你可以用 DEFAULT 和 ON UPDATE 属性的任意组合来定义它，如下所示。

- 如果给出了 DEFAULT CURRENT_TIMESTAMP 短语，该数据列将具备自动初始化属性。如果还给出了 ON UPDATE CURRENT_TIMESTAMP 短语，它还将具备自动更新属性。
- 如果省略了这两个属性，MySQL 将把它定义为 DEFAULT CURRENT_TIMESTAMP 和 ON UPDATE CURRENT_TIMESTAMP。
- 如果你在 DEFAULT *constant_value* 短语里给出了一个常数值，该数据列将不具备自动初始化属性。但如果你给出了 ON UPDATE CURRENT_TIMESTAMP 短语，它将具备自动更新属性。
- 如果只给出了 ON UPDATE CURRENT_TIMESTAMP、没有给出 DEFAULT 部分，该数据列的默认值将是 0，并且具备自动更新属性。

如果你想让数据表里的第二个或更靠后的某个 TIMESTAMP 数据列具备自动初始化或自动更新属性，就必须在定义第一个 TIMESTAMP 数据列的时候明确地给出一个 DEFAULT *constant_value* 属性且不能给出 ON UPDATE CURRENT_TIMESTAMP 属性。只有这样，你才能在定义其他 TIMESTAMP 数据列时使用 DEFAULT CURRENT_TIMESTAMP 或 ON UPDATE CURRENT_TIMESTAMP（或同时使

用它们两个)。

如果不想让某个 TIMESTAMP 数据列具备自动初始化或自动更新属性，在插入或修改语句里把你想保存到该数据列的值明确地写出来即可。比如说，如果你不想让修改其他数据列的操作改变 TIMESTAMP 数据列的值，把后者设置为它的当前值就行了。

在 TIMESTAMP 数据列的定义里还可以包含 NULL 或 NOT NULL。默认设置是 NOT NULL，其效果是当你明确地把 TIMESTAMP 数据列设置为 NULL 时，MySQL 会把它设置为当前的时间戳值（插入操作和修改操作都会如此)。如果你给某个 TIMESTAMP 数据列定义了 NULL 属性，当你把该数据列设置为 NULL 时，MySQL 将把 NULL 而不是当前时间戳值存入该数据列。

如果你想让数据表里有这样一个数据列：在插入新数据行时被设置为当前时间戳值，以后就再也不变了。有两个办法可以让你达到目的，如下所示。

- 使用一个 TIMESTAMP 数据列，它的定义如下所示，不带 ON UPDATE 属性：

  ```
  ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP
  ```

  当你创建一个新数据行时，在 INSERT 语句里把这个 TIMESTAMP 数据列设置为 NULL 或干脆省略它将使得它被初始化为当前的时间戳值。在以后的修改操作里，这个数据列的值将保持不变——除非你明确地改变了它。
- 使用一个 DATETIME 数据列。在创建新数据行时，把该数据列初始化为 NOW()；在以后的修改操作中不再去修改它。

如果想把数据行的创建时间和最近一次修改时间都记录下来，需要使用两个 TIMESTAMP 数据列：

```
CREATE TABLE t
(
  t_created  TIMESTAMP DEFAULT 0,
  t_modified TIMESTAMP DEFAULT CURRENT_TIMESTAMP
                       ON UPDATE CURRENT_TIMESTAMP
  ... other columns ...
);
```

此后，当你插入一个新数据行时，请把这两个 TIMESTAMP 数据列都设置为 NULL，这将使它们都设置为当前（插入新数据行时）的时间戳值；当你修改一个现有的数据行时，这两个 TIMESTAMP 数据列都不要去碰，t_modified 数据列自动更新为当前的（修改发生时）时间戳值——当然，前提是其他数据列的值确实发生了变化。

3. YEAR数据类型

YEAR 是一种单字节的数据类型，既能节约存储空间，又能提高处理效率，非常适用于只会用到年份值的场合。在声明 YEAR 数据列时，可以为它设定一个显示宽度 M，只能是 4 或 2。如果没有在 YEAR 数据列的声明定义里对 M 值进行设定，其默认值将是 4。YEAR(4)的取值范围是 1901 年到 2055 年；YEAR(2)的取值范围是 1970 年到 2069 年，但只显示最后两位数字。如果你只需用到日期值里的年份数字，比如出生年份、公司创建年份等，就应该使用 YEAR 类型。如果只需要年份值，在各种日期/时间类型中，YEAR 类型的存储空间占用量是最小的。

TINYINT 类型的存储空间占用量与 YEAR 类型一样（都是一个字节)，但它们的取值范围却完全不一样。如果你想用一个整数类型来覆盖 YEAR 类型所能表示的年份区间，那就得选用 SMALLINT——空间占用量将翻一番。因此，如果需要表示的年份都落在 YEAR 类型能够表示的年份区间里，与使用 SMALLINT 数据列的情况相比，使用 YEAR 数据列将节约一半的存储空间。使用 YEAR 类型的另一个好

处是 MySQL 可以根据它的“年份猜测规则”自动地把两位数字的年份值转换为四位数字。比如说，97 和 14 将分别被转换为 1997 和 2014。不过，需要提醒大家注意的是，如果把数值 00 插入到一个 YEAR(4)数据列里，转换结果将是 0000 而不是 2000。因此，如果你想让 00 转换为 2000，就一定要使用它的字符串形式'0'或'00'。

4. 日期/时间数据类型的属性

以下内容适用于除 TIMESTAMP 以外所有日期/时间类型。

- 可以给它们加上通用的 NULL 或 NOT NULL 属性。如果没有设定，MySQL 将默认地给它们加上 NULL 属性。
- 还可以用 DEFAULT 子句来设定一个默认值。如果没有设置默认值，MySQL 将按照 3.2.3 节里的有关规则为它们挑选一个。

需要提醒大家注意的是，因为默认值必须是一个常数，所以不能通过 NOW()函数把“当前日期和时间”设置为 DATETIME 数据列的默认值。如果你想得到这种效果，有 3 种办法可供选择：在创建新数据行时把 DATETIME 数据列初始化为 NOW()函数的返回值；使用一个 TIMESTAMP 数据列；安排一个触发器把该数据列初始化为适当的值，详见 4.3 节。

TIMESTAMP 数据列的情况比较特殊，简单地说，只有数据表里的第一个 TIMESTAMP 数据列的默认值才会是当前的日期和时间，其余数据列（如果有的话）的默认值将是该类型的“零值”。MySQL 用来为 TIMESTAMP 数据列确定默认值的完整规则相当复杂，详情请参阅“TIMESTAMP 数据类型”一节。

5. 日期/时间值的使用

MySQL 能够识别和使用多种格式的日期/时间值，包括它们的字符串形式和数值形式。表 3-16 列出了 MySQL 所支持的各种日期/时间格式。

表3-16　日期/时间类型的输入格式

类　　型	允许使用的输入格式
DATETIME, TIMESTAMP	'*CCYY-MM-DD hh:mm:ss*' '*YY-MM-DD hh:mm:ss*' '*CCYYMMDDhhmmss*' '*YYMMDDhhmmss*' *CCYYMMDDhhmmss* *YYMMDDhhmmss*
DATE	'*CCYY-MM-DD*' '*YY-MM-DD*' '*CCYYMMDD*' '*YYMMDD*' *CCYYMMDD* *YYMMDD*
TIME	'*hh:mm:ss*' '*hhmmss*' *hhmmss*
YEAR	'*CCYY*' '*YY*' *CCYY* *YY*

对于那些没有世纪部分（*CC*）的日期/时间格式，MySQL 将根据第 6 小节里的规则来进行解释。如果是带分隔符的字符串格式，那就不必用“-”和“:”来分隔日期或时间值中的各个部分，任何一种标点符号都可以用作日期/时间值中的分隔符。MySQL 是根据上下文而不是分隔符来解释日期/时间值的。比如说，虽然人们习惯于使用“:”来分隔时间值中的各个部分，但在需要使用日期值的场合，MySQL 并不会把一个用“:”分隔的值解释为时间值。此外，如果你使用的是带分隔符的字符串格式，小于 10 的月份、日期、小时、分钟或秒就用不着写成两位数字。比如说，下面这些日期/时间值就都是等价的：


```
'2012-02-03 05:04:09'
'2012-2-03 05:04:09'
'2012-2-3 05:04:09'
'2012-2-3 5:04:09'
'2012-2-3 5:4:09'
'2012-2-3 5:4:9'
```

对于日期/时间值里的前导的零，MySQL 对字符串格式和数值格式的解释方法是不同的。比如说，字符串'001231'将被看做是一个 6 位数字的值，并被解释为 DATE 类型的'2000-12-31'或 DATETIME 类型的'2000-12-31 00:00:00'。可是，数值 001231 却会被看做是 1231，对它的解释就有点复杂了。在这类场合，为了避免出现预想不到的后果，还是使用它的字符串形式'001231'或完整的数值形式（如果你想输入的是一个 DATE 值，就应该把它写成 20001231；如果你想输入的是一个 DATETIME 值，就应该把它写成 200012310000）比较好。

在一般情况下，DATE、DATETIME 和 TIMESTAMP 类型上的赋值操作可以交叉互用。但你必须清楚以下几个限制条件。

- 如果把一个 DATETIME 或 TIMESTAMP 值赋值给一个 DATE 数据列，它里面的时间值将被删除。
- 如果把一个 DATE 值赋值给一个 DATETIME 或 TIMESTAMP 数据列，MySQL 将自动补足一个时间零值（'00:00:00'）。
- 不同的日期/时间类型有不同的取值范围。尤其是 TIMESTAMP 类型，它的取值范围只是从 1970 到 2069 而已。因此，如果你试图把一个早于 1970 年或晚于 2069 年的 DATETIME 值赋值给一个 TIMESTAMP 数据列，就不能期待会得到一个合理的结果。

MySQL 有很多用来对日期/时间值进行处理的函数。有关这些函数的详细情况请参阅附录 C。

6. 确定日期值中的年份

对于那些带有年份值的日期/时间类型（DATE、DATETIME、TIMESTAMP、YEAR），MySQL 能够自动地根据以下规则把两位数字的年份值转换为四位数字：

- 年份值 00 到 69 将被转换为 2000 到 2069。
- 年份值 70 到 99 将被转换为 1970 到 1999。

把各种两位数的年份值赋给一个 YEAR 数据列再把它们检索出来，你就能看到上述转换规则的实际效果了。你还发现一些你也许不曾留意的东西，如下所示：

```
mysql> CREATE TABLE y_table (y YEAR);
mysql> INSERT INTO y_table VALUES(68),(69),(99),(00),('00');
mysql> SELECT * FROM y_table;
+------+
| y    |
+------+
```

```
| 2068 |
| 2069 |
| 1999 |
| 0000 |
| 2000 |
+------+
```

请注意，00被转换为0000而不是2000。这是因为数值00就等于0，而0又是YEAR类型的一个合法取值。如果你插入的是一个数值0，其结果就将是0000。因此，如果你想使用一个没有世纪部分的值得到表示2000年的结果，就必须使用字符串形式'0'或'00'。如果想让MySQL把YEAR数据列里插入的值看做是字符串而不是数值，就应该使用CAST(*value* AS CHAR)函数。无论*value*是一个字符串还是一个数值，这个函数都将返回一个字符串。

需要提醒大家的是，用来把两位数字的年份值转换为四位数字的转换规则只是MySQL一种比较合理的猜测。MySQL并不知道你的两位数（后两位）年份到底指的是哪一年。因此，虽然MySQL的年份转换规则在许多场合都很适用，万一产生的结果与你的预期不一致，最好还是提供一个不可能导致歧义的四位数年份值。比如说，如果你想把自 18 世纪以来的美国总统的出生日期和去世日期录入到president数据表里去，就必须使用四位数字的年份值。这些值跨越了好几个世纪，让MySQL根据两位数的年份值去猜测它们属于哪个世纪绝非明智之举。

3.2.7　空间数据类型

坐标值用来表示点、线和多边形。MySQL里的空间数据类型是按照OpenGIS标准实现的，可以在Open Geospatial Consortium（开放地理信息联盟）的官方网站（http://www.opengeospatial.org/）上查到这份标准。表3-17列出了MySQL支持的空间数据类型。

表3-17　空间数据类型

类型名称	说　　明
GEOMETRY	任意类型的坐标值
POINT	一个点（一组X、Y坐标值）
LINESTRING	一条曲线（一个或多个POINT值）
POLYGON	一个多边形
GEOMETRYCOLLECTION	一个由GEOMETRY值构成的集合
MULTILINESTRING	一个由LINESTRING值构成的集合
MULTIPOINT	一个由POINT值构成的集合
MULTIPOLYGON	一个由POLYGON值构成的集合

不同的存储引擎对空间类型的支持程度是不同的。MyISAM存储引擎对空间数据类型的支持最完善，其他存储引擎（如InnoDB、NDB和ARCHIVE等）提供的支持就很有限了。比如说，在MyISAM数据表里，坐标值既可以用来创建SPATIAL索引，也可以用来创建非SPATIAL索引（使用INDEX、UNIQUE 或 PRIMARY KEY）。支持空间数据类型的其他存储引擎只能使用非 SPATIAL 索引（不包括ARCHIVE存储引擎，它根本不支持对空间数据列编制索引）。此外，分区数据表不能包含空间数据列。

包括在一个SPATIAL索引里的空间数据列不能使用NULL值来表示“数据列里没有东西”，因为SPATIAL索引不允许用NULL值。根据具体情况，可以用一个空值（零长度的值）来代替。

MySQL 可以识别和使用 3 种坐标值格式。其中一种是 MySQL 用来在数据表里保存坐标值的内部格式。另外两种是 WKT（Well-Known Text）和 WKB（Well-Known Bianry）格式，它们是用文本字符串和二进制数据来表示坐标值的行业标准，OpenGIS 标准对 WKT 和 WKB 格式的语法进行了定义。比如说，如果使用 WKT 格式，一个由坐标值 *x* 和 *y* 给定的 POINT 值将被写成如下所示的字符串：

```
'POINT(x y)'
```

请注意，坐标值 *x* 和 *y* 之间没有逗号或其他分隔符。在列出多组坐标时，需要用逗号用来隔开各组坐标。下面的字符串代表一个 LINESTRING 值，它由好几个点构成：

```
'LINESTRING(10 20, 0 0, 10 20, 0 0)'
```

更复杂的值有更复杂的表示形式。下面这个 POLYGON 有一个矩形外边界和一个三角形内边界：

```
'POLYGON((0 0, 100 0, 100 100, 0 100, 0 0),(30 30, 30 60, 45 60, 30 30))'
```

因为坐标值可能很复杂，所以对它们的处理操作大都需要通过调用相应的函数来进行。坐标函数的数量很多，包括用来把一种格式转换为另一种格式的函数（详见附录 C）。

下面的例子演示了坐标数据的几种用法：

```
mysql> CREATE TABLE pt_tbl (p POINT);
mysql> INSERT INTO pt_tbl (p) VALUES
    ->   (POINTFROMTEXT('POINT(0 0)')),
    ->   (POINTFROMTEXT('POINT(0 50)')),
    ->   (POINTFROMTEXT('POINT(100 100)'));
mysql> CREATE FUNCTION dist (p1 POINT, p2 POINT)
    ->   RETURNS FLOAT DETERMINISTIC
    ->   RETURN SQRT(POW(X(p2)-X(p1),2) + POW(Y(p2)-Y(p1),2));
mysql> SET @ref_pt = POINTFROMTEXT('POINT(0 0)');
mysql> SELECT ASTEXT(p), dist (p, @ref_pt) AS dist FROM pt_tbl;
+----------------+-----------------+
| ASTEXT(p)      | dist            |
+----------------+-----------------+
| POINT(0 0)     |               0 |
| POINT(0 50)    |              50 |
| POINT(100 100) | 141.42135620117 |
+----------------+-----------------+
```

这个例子进行了以下操作。

(1) 创建了一个数据表，其中包含一个空间数据列。

(2) 把一些 POINT 值填充到数据表里，用 POINTFROMTEXT()函数把 WKT 格式的坐标数据转换为 MySQL 的内部格式。

(3) 创建了一个存储函数来计算两点之间的距离，这里使用了 X()和 Y()函数来提取各个点的坐标。

(4) 计算数据表里的每个点与给定参考点之间的距离。

3.3 MySQL 如何处理非法数据值

在过去，MySQL 在处理数据时的基本原则是“垃圾进、垃圾出”。换句话说，你给 MySQL 什么数据，它就保存什么数据，但如果你在存储数据时没有把好关，你从中检索出来的就不一定是你希望的样子了。从 MySQL 5.0.2 版开始，MySQL 增加了几种 SQL 模式，它们可以让数据库拒绝接受“坏”

数据并抛出一个错误。接下来我们将先讨论 MySQL 对非法数据的默认处理措施，然后讨论启用各种 SQL 模式会对数据处理产生哪些影响。

在默认的情况下，MySQL 按照以下规则处理“数据越界”和其他非正常数据。

- 对于数值数据列或 TIME 数据列，超出合法范围的值将被截短到最近的取值范围边界，然后把结果值存入数据库。
- 对于字符串数据列（不包括 ENUM 和 SET），太长的字符串将被截短到数据列的最大长度。
- 对 ENUM 和 SET 数据列的赋值操作取决于在数据列定义里给出的合法取值列表。如果你赋值给某个 ENUM 数据列的值不是合法成员，MySQL 将把“出错”成员（也就是与零值成员相对应的空字符串）赋值给该数据列。如果你赋值给某个 SET 数据列的值包含非合法子字符串，MySQL 将删除那些子字符串而只把剩下来的东西赋值给该数据列。
- 对于日期和时间数据列，非法值将被转换为该类型的“零值”（参见表 3-15）。

如果在执行 INSERT、REPLACE、UPDATE、LOAD DATA 和 ALTER TABLE 等语句时发生上述转换，MySQL 将生成一条警告消息。在这几种语句执行完毕之后，你可以用 SHOW WARNINGS 语句去查看警告消息的内容。

如果需要在插入或更新数据时进行更严格的检查，可以启用以下两种 SQL 模式之一：

```
mysql> SET sql_mode = 'STRICT_ALL_TABLES';
mysql> SET sql_mode = 'STRICT_TRANS_TABLES';
```

对于支持事务的数据表，这两种模式是一样：如果发现某个值非法或缺失，MySQL 将抛出一个错误，那条语句将停止执行并回滚，就像它从未执行过一样。对于不支持事务的数据表，这两种模式有以下效果。

- 在这两种模式下，如果在插入或修改第一个数据行时就发现某个值非法或缺失，抛出一个错误，语句停止执行，就像从未执行过一样，这和支持事务的数据表的行为很相似。
- 如果在插入或修改第 *n* 个（*n*>1）数据行时才发现某个值非法或缺失，就已经有一些数据行被修改了。这两种严格模式将决定是在此刻停止语句的执行还是让它继续执行，如下所示。
    - 在 STRICT_ALL_TABLES 模式下，这时将抛出一个错误，停止语句的执行。因为已经有一些数据行被修改了，但还有一些数据行没被修改，所以这将导致“部分更新”问题。
    - 在 STRICT_TRANS_TABLES 模式下，如果能够获得支持事务的数据表那样的效果，对不支持事物的数据表的语句将停止执行。这个条件只在错误发生在第一个数据行时才能得到满足，如果错误发生在第 *n* 个（*n*>1）数据行上，就已经有一些数据行被修改了。既然是不支持事务的数据表，那些修改就无法撤销，而 MySQL 将继续执行该语句以避免“部分更新”。对于每一个非法值，MySQL 将按刚才介绍的规则把它转换为最接近的合法值。对于缺失的值，MySQL 将根据其数据类型为它挑选一个隐式默认值，详见 3.2.3 节。

严格模式并不是 MySQL 能进行的最严格的检查。你还可以通过以下模式之一或全部来对输入数据进行更严格的检查。

- ERROR_FOR_DIVISION_BY_ZERO：如果在严格模式下遇到以零为除数的情况，拒绝数据进入数据库。（如果不在严格模式下，MySQL 将生成一条警告并插入 NULL 值。）
- NO_ZERO_DATE：在严格模式下，拒绝“零”日期值进入数据库。
- NO_ZERO_IN_DATE：在严格模式下，拒绝月或日部分是零的不完整日期值进入数据库。

比如说，如果你想为所有的存储引擎启用严格模式并检查“零除数”错误，请把 SQL 模式设置

成下面这样：

```
mysql> SET sql_mode = 'STRICT_ALL_TABLES,ERROR_FOR_DIVISION_BY_ZERO';
```

如果想启用严格模式和所有的附加检查，最简单的办法是启用 TRADTIONAL 模式：

```
mysql> SET sql_mode = 'TRADITIONAL';
```

TRADTIONAL 模式的含义是“启用两种严格模式，再加上所有其他检查”。这与其他“传统的”SQL DBMS 系统在数据检查方面的行为更相似。

你还可以选择性地弱化严格模式的某些方面。如果你启用了 ALLOW_INVALID_DATES SQL 模式，MySQL 将不对日期部分做全面的检查，只要求月份值落在 1 到 12 之间、日落在 1 到 31 之间就行了（这意味着'2000-02-30'和'2000-06-31'这样的非法值将被认为是合法的）。制止错误的另一个办法是在 INSERT 或 UPDATE 语句里使用 IGNORE 关键字，因非法值而导致的错误将被弱化为一个警告。

MySQL 已经在输入数据的检查方面准备了各种各样的选项，你可以根据具体需要灵活选用。

3.4 序列

很多应用都需要使用一些独一无二的编号来作为标识，如会员号、抽样编号、顾客编号、程序漏洞报告或故障报告表编号等。

MySQL 这种独一无二的编号机制是通过数据列的 AUTO_INCREMENT 属性而自动生成一组序列编号的办法来实现的。MySQL 支持的多种数据表类型对 AUTO_INCREMENT 数据列的处理办法是不一样的。因此，你不仅需要掌握 AUTO_INCREMENT 机制的基本概念，还必须熟悉这种机制在各种数据表类型中的差异。为帮助大家更好地掌握这一机制并避免落入各种陷阱，本节将介绍 AUTO_INCREMENT 数据列的一般工作原理和针对特定存储引擎的工作原理。此外，本节还会介绍几种不使用 AUTO_INCREMENT 数据列的序列编号生成办法。

3.4.1 通用AUTO_INCREMENT属性

AUTO_INCREMENT 数据列必须按照以下条件定义。

- 每个数据表只能有一个数据列具备 AUTO_INCREMENT 属性，而且它应该有一种整数数据类型。（AUTO_INCREMENT 也支持浮点类型的数据列，但需要那么做的情况极少。）
- 必须给该数据列添加索引。使用一个 PRIMARY KEY 或 UNIQUE 索引的情况最常见，但使用一个非唯一索引的情况也是允许的。
- 必须给该数据列添加一个 NOT NULL 约束条件。即使你没有明确地做出这样的声明，MySQL 也会自动地把该数据列设置为 NOT NULL。

在创建出来之后，AUTO_INCREMENT 数据列将具备以下行为特点。

- 把 NULL 值插入一个 AUTO_INCREMENT 数据列将使 MySQL 自动生成下一个序列编号并把该值插入该数据列。

  AUTO_INCREMENT 序列通常从 1 开始依次单步递增，这意味着连续插入某个数据表的数据行的序号值将是 1、2、3，等等。在某些场合，根据具体使用的存储引擎，可以明确地对下一个序号进行设置或重新设置，或者是重复使用已经从序列顶端被删除的序号值。
- 最近生成的序号值可以通过调用 LAST_INSERT_ID()函数获得。这就使你可以在后续的语句里引用 AUTO_INCREMENT 值，即便你不知道这个值到底是什么也不要紧。如果在建立本次连

接以后还没有生成过 AUTO_INCREMENT 值，LAST_INSERT_ID()函数将返回 0。

LAST_INSERT_ID()函数只能返回本次连接服务器以后最近一次生成的 AUTO_INCREMENT 值。换句话说，它不受和其他客户相关联的 AUTO_INCREMENT 活动的影响。你可以生成一个序号值，然后在同一次连接的后续语句里调用 LAST_INSERT_ID()函数检索它，即使还有其他客户在此期间生成了它们自己的序号值也不会出问题。

一次插入多个数据行的 INSERT 语句将生成多个 AUTO_INCREMENT 值，LAST_INSERT_ID()函数将只返回它们当中的第一个。

如果为支持延迟插入功能的存储引擎使用了 INSERT DELAYED 语句，AUTO_INCREMENT 值将在数据行被实际插入时才会生成。在这种情况下，通过调用 LAST_INSERT_ID()函数来返回序号值的办法将不再可靠。

- 在插入一个数据行时没有明确地为AUTO_INCREMENT 数据列给出一个值与把 NULL 值插入该数据列的效果完全一样。如果 ai_col 是一个 AUTO_INCREMENT 数据列，下面两条语句将是等效的：

```
INSERT INTO t (ai_col,name) VALUES(NULL,'abc');
INSERT INTO t (name) VALUES('abc');
```

- 在默认的情况下，把 0 插入一个 AUTO_INCREMENT 数据列与插入 NULL 值的效果相同。但如果你启用了 NO_AUTO_VALUE_ON_ZERO SQL 模式，插入的 0 将被存储为 0 而不是下一个序号值。
- 如果你在插入某个数据行的时候为一个有着唯一索引的 AUTO_INCREMENT 数据列给出了一个既不是 NULL 也不是零的值，将发生以下两种情况之一。如果已经存在一个用到了那个值的数据行，就会发生键重复错误。如果不存在一个用到了那个值的数据行，新数据行将被插入，其 AUTO_INCREMENT 数据列的值将被设置为你给出的值。如果该值大于当前预期的下一个序号值，序列将被重置，随后插入的数据行将在该值的基础上继续编号。换句话说，你可以通过插入一个其序号值大于当前计数器值的数据行的办法让计数器“跳”过一个区间。

  跳过一些编号会导致编号序列里出现断裂带，你可以利用这一点来生成一组从大于 1 的某个数开始的编号。假设你创建了一个数据表，里面有一个 AUTO_INCREMENT 数据列，但你想从 1 000 而不是 1 开始编号。为此，你先得插入一条“假”行，把它的 AUTO_INCREMENT 数据列的值设置为 999。这样，以后插入的记录就会从 1 000 开始编号了。等那条“假”行完成其使命之后，就可以删掉了。

  为什么要让序列编号从大于 1 的数开始呢？一种理由是为了使全体编号都由相同个数的数字组成。比如说，如果你正在生成顾客 ID 编号，并且预计顾客不会超过 100 万个，就可以让这些编号从 1 000 000 开始。这样，你可以添加 100 万多个顾客记录，每位顾客的 ID 编号就都将是一个 7 位数字（至少在顾客人数超过 9 999 999 个之前是这样的）。
- 对于某些存储引擎，从序列顶端删除的值会被重复使用。在这种情况下，如果你刚删除的数据行包含着 AUTO_INCREMENT 数据列的最大值，这个值将在你生成一个新值时被重新使用。这一特性的隐含效果之一是如果你删除了某个数据表里的所有数据行，那么所有的值都将被重复使用，序列将从头从 1 开始。
- 如果用 UPDATE 命令把 AUTO_INCREMENT 数据列里的值设置成一个正被其他记录使用着的编号，并且数据列有唯一的索引，就会看到一条“键字重复”的出错信息。如果你新设置的编号值大于任何一个现有编号，以后再插入的新记录将从你设置的新编号值开始继续编号。如

果你通过把 0 赋值给它来更新该数据列，它将被设置为 0（不管是否已经启用了 NO_AUTO_VALUE_ON_ZERO SQL 模式）。

- 如果你使用 REPLACE 命令根据 AUTO_INCREMENT 数据列的值去更新一个数据行，其 AUTO_INCREMENT 值将保持不变。如果你使用 REPLACE 命令根据其他的 PRIMARY KEY 或 UNIQUE 索引的值去更新一个数据行，那么，如果你把其 AUTO_INCREMENT 数据列设置为 NULL，或是把它设置为 0 且没有启用 NO_AUTO_VALUE_ON_ZERO 的话，它将被更新为一个新的序号值。

3.4.2 与特定存储引擎有关的AUTO_INCREMENT属性

刚才介绍的通用 AUTO_INCREMENT 属性是理解各种存储引擎独有的序列行为的基础。绝大多数引擎所实现的行为在绝大多数方面与上面的描述相一致，所以请大家在阅读下面的内容时把刚才的讨论记在脑子里。

1. MyISAM数据表里的AUTO_INCREMENT数据列

MyISAM 数据表对序列编号的处理是最灵活的。MyISAM 存储引擎有以下 AUTO_INCREMENT 特征。

- MyISAM 数据表里的序列是单调的。一个自动生成的序列，其编号值将严格地依次递增而不会被再次使用（如果你删除数据行）。如果当前的最大编号是 143 而你又删除了包含这个编号的数据行，MySQL 生成的下一个编号将是 144。不过，有以下两种例外情况。
- 如果使用 TRUNCATE TABLE 命令清空了一个数据表，计数器将被重置为从 1 开始。
- 如果用一个复合索引在数据表里生成了多个序列，从序列顶端删除的值将被重复使用。（这个技巧马上就会讨论到。）
- 如果你没有让编号从一个较大的值开始递增，MyISAM 序列将默认从 1 开始编号。在 MyISAM 数据表里，可以在 CREATE TABLE 语句里通过 AUTO_INCREMENT = *n* 选项为序列编号明确地设定一个初始值。在下面这条语句所创建的 MyISAM 数据表里，那个名为 seq 的 AUTO_INCREMENT 数据列将从 1 000 000 开始编号：

  ```
  CREATE TABLE mytbl
  (
    seq INT UNSIGNED NOT NULL AUTO_INCREMENT,
    PRIMARY KEY (seq)
  ) ENGINE = MYISAM AUTO_INCREMENT = 1000000;
  ```

  每个 MyISAM 数据表最多只能有一个 AUTO_INCREMENT 数据列。因此，即使数据表里有很多个数据列，出现在 CREATE TABLE 语句最末尾的 AUTO_INCREMENT = *n* 选项也不会引起歧义。
- 你可以用 CREATE TABLE 命令改变 MyISAM 数据表里当前序列编号编号值。比如说，如果序列的当前编号是 1000，那么下面这条语句就将使下一个编号从 2000 开始：

  ```
  ALTER TABLE mytbl AUTO_INCREMENT = 2000;
  ```

  如果你刚刚删除了编号最大的那行，那你完全可以再次使用那个编号。具体做法是：把编号计数器尽可能地往低处设置，而这将使下一个编号只比现有编号的最大值多 1。如下所示：

  ```
  ALTER TABLE mytbl AUTO_INCREMENT = 1;
  ```

  你不能使用 AUTO_INCREMENT 选项把当前计数值设置得比数据表里现有的最大计数值还低。比如说，如果某个 AUTO_INCREMENT 数据列包含值 1 和 10，就算用 AUTO_INCREMENT = 5 去设置计数器，自动生成的下一个序号值也将是 11。

MyISAM 存储引擎支持使用复合（多数据列）索引在同一数据表里创建多个相互独立的序列。具体用法是：为数据表创建一个由多个数据列组成的 PRIMARY KEY 或 UNIQUE 索引，并把一个 AUTO_INCREMENT 数据列包括在这个索引里作为它的最后一个数据列。对应于该索引中最左边的数据列（或最左边的几个数据列）构成的每一个不同的键，AUTO_INCREMENT 数据列将生成一组彼此互不干扰的序列值。比如说，用一个名为 bugs 的数据表来同时记录多个软件项目的 bug 报告，下面是这个数据表的定义：

```
CREATE TABLE bugs
(
  proj_name   VARCHAR(20) NOT NULL,
  bug_id      INT UNSIGNED NOT NULL AUTO_INCREMENT,
  description VARCHAR(100),
  PRIMARY KEY (proj_name, bug_id)
) ENGINE = MYISAM;
```

proj_name 数据列用来存放项目的名称，description 数据列用来存放对 bug 的描述。bug_id 数据列用来存放 bug 的编号，它是一个 AUTO_INCREMENT 数据列。我们还创建了一个与 proj_name 数据列相关联的索引，这样，我们就能为各个项目分别生成一个互不干扰的序列编号了。接下来把下面 5 条数据行插入到数据表里，其中有 3 条是 SuperBrowser 项目的 bug，有 2 条是 SpamSquisher 项目的 bug：

```
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SuperBrowser','crashes when displaying complex tables');
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SuperBrowser','image scaling does not work');
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SpamSquisher','fails to block known blacklisted domains');
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SpamSquisher','fails to respect whitelist addresses');
mysql> INSERT INTO bugs (proj_name,description)
    -> VALUES('SuperBrowser','background patterns not displayed');
```

下面是我们发出的查询和它的结果：

```
mysql> SELECT * FROM bugs ORDER BY proj_name, bug_id;
+--------------+--------+------------------------------------------+
| proj_name    | bug_id | description                              |
+--------------+--------+------------------------------------------+
| SpamSquisher |      1 | fails to block known blacklisted domains |
| SpamSquisher |      2 | fails to respect whitelist addresses     |
| SuperBrowser |      1 | crashes when displaying complex tables   |
| SuperBrowser |      2 | image scaling does not work              |
| SuperBrowser |      3 | background patterns not displayed        |
+--------------+--------+------------------------------------------+
```

只要 bug_id 值是按项目分组排列的，哪行在先哪行在后倒无关紧要。你不是非要输完某个项目的所有行才能输另一个项目的数据行。

如果你在这个数据表里用一个复合索引创建了多个序列，从各序列顶端删除的值将可以重复使用。这种现象与 MyISAM 数据表不重复使用序号值的行为是不同的。

2. MEMORY数据表中的AUTO_INCREMENT数据列

MEMORY 存储引擎有以下 AUTO_INCREMENT 特性。

- 允许在 CREATE TABLE 语句里用 AUTO_INCREMENT = *n* 数据表选项来设置初始序号值，还允许在数据表创建之后用 ALTER TABLE 语句改变序号值。
- 从序列顶端删除的值通常不再重复使用。但如果用 TRUNCATE TABLE 清空了数据表的话，序列将被重置为重新从 1 开始编号。
- 不能使用复合索引在同一个数据表里生成多个相互独立的序列。

3. InnoDB数据表中的AUTO_INCREMENT数据列

InnoDB 存储引擎有以下 AUTO_INCREMENT 特性。

- 从 MySQL 5.0.3 开始，允许在 CREATE TABLE 语句里用 AUTO_INCREMENT = *n* 数据表选项来设置初始序号值，还允许在数据表创建之后用 ALTER TABLE 语句改变序号值。
- 从序列顶端删除的值通常不再重复使用。但如果用 TRUNCATE TABLE 清空了数据表的话，序列将被重置为重新从 1 开始编号。重复使用序号值的情况在满足以下条件时也可以发生：在首次为一个 AUTO_INCREMENT 数据列生成一个序号值时，InnoDB 将以该数据列里的当前最大值加上 1 后作为它生成的新序号值（如果数据表此前是空的，新序号值将是 1）。InnoDB 在内存里维护着这个计数器以生成后续的序号值，该计数器并未存储在数据表本身的内部。这意味着，如果你从这个序列的顶端删除了一些值以后重新启动服务器，你删除过的那些值将被重复使用。重新启动服务器还将取消在 CREATE TABLE 或 ALTER TABLE 语句里使用 AUTO_INCREMENT 数据表选项的效果。
- 如果生成 AUTO_INCREMENT 值的交易被回滚，序列中可能会出现“断裂带”。
- 复合索引不能用来在一个表里生成多个独立的序列。

3.4.3 使用AUTO_INCREMENT数据列时的要点

在使用 AUTO_INCREMENT 数据列时，为避免陷入不必要的麻烦，请记住以下几个要点。

- 虽然我们常提到 AUTO_INCREMENT 数据列，但它并不是一种数据列类型，只是数据列类型的一个属性。进一步讲，AUTO_INCREMENT 是一种只适用于整数类型或浮点类型的属性。MySQL 3.23 旧版本对这一点要求得并不是很严格。它们允许你给诸如 CHAR 这样的类型也定义 AUTO_INCREMENT 属性，但只有整数类型或浮点类型上的 AUTO_INCREMENT 数据列才能正确地工作。
- AUTO_INCREMENT 机制的基本用途是生成一个正整数序列。MySQL 不支持在 AUTO_INCREMENT 数据列里使用非正数，所以我们完全可以把 AUTO_INCREMENT 数据列定义为 UNSIGNED 类型。对整数数据列而言，使用 UNSIGNDE 的好处是：在到达数据类型的最大可取值之前，你有两倍的序列编号可用。
- 千万不要自以为是地认为把 AUTO_INCREMENT 添加到数据列声明里就能得到无穷无尽的序列编号。这种想法是错误的。AUTO_INCREMENT 序列要受制于基本数据列类型的取值范围。比如说，如果你使用的是一个 TINYINT 数据列，那么序列编号的最大值就将是 127。一旦达到这个上限，就会因“键字重复”错误而使操作失败。如果你使用的是 TINYINT UNSIGNED 数据列，序列编号的上限值就将是 255。
- 使用 TRUNCATE TABLE 语句清除某个数据表的内容将导致该数据表中的计数序列被重置为重新从 1 开始计数，这对通常不重用 AUTO_INCREMENT 值的几种存储引擎来说也不例外。序列重置是因为 MySQL 对一个完备的数据表删除操作进行优化时的方式。只要有可能，它就会快

速丢弃全部的数据行和索引，从零开始重新创建该数据表，而不是一行一行删除，而这将导致序列编号信息丢失。如果你想删除所有的数据行但保留序列信息，可以用一条 WHERE 子句的 DELETE 语句去抑制这种优化，迫使 MySQL 为每个数据行求值并因此删除每一行：

```
DELETE FROM tbl_name WHERE TRUE;
```

3.4.4　使用AUTO_INCREMENT机制时的注意事项

本节讨论使用 AUTO_INCREMENT 数据列时的一些有用技巧。

1. 给数据表增加一个序列编号数据列

假设你已经创建了一个数据表并在里面存入了一些信息：

```
mysql> CREATE TABLE t (c CHAR(10));
mysql> INSERT INTO t VALUES('a'),('b'),('c');
mysql> SELECT * FROM t;
+------+
| c    |
+------+
| a    |
| b    |
| c    |
+------+
```

现在，你要给这个数据表增加一个序列编号数据列。于是，用一条 ALTER TABLE 语句给它增加一个 AUTO_INCREMENT 数据列，而它的声明定义应该与你用 CREATE TABLE 语句创建这样的数据列时使用的一样，如下所示：

```
mysql> ALTER TABLE t ADD i INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY;
mysql> SELECT * FROM t;
+------+---+
| c    | i |
+------+---+
| a    | 1 |
| b    | 2 |
| c    | 3 |
+------+---+
```

请注意，MySQL 自动完成了 AUTO_INCREMENT 数据列里的序列编号赋值工作，根本用不着你亲自去做这件事。

2. 重新编排现有的序列编号

如果你的数据表里已经有了一个 AUTO_INCREMENT 数据列，但你现在想对它重新编号以消除因删除数据行而在序列中产生的断裂带。最简单的办法是：先删除该数据列，然后再重新添加它。MySQL 会在重新添加该数据列时自动完成序列编号的赋值工作，就像前面那个例子里一样。

先创建一个数据表 t，它里面有一个 AUTO_INCREMENT 数据列 i：

```
mysql> CREATE TABLE t (c CHAR(10), i INT UNSIGNED AUTO_INCREMENT
    -> NOT NULL PRIMARY KEY);
mysql> INSERT INTO t (c)
    -> VALUES('a'),('b'),('c'),('d'),('e'),('f'),('g'),('h'),('i'),('j'),('k');
mysql> DELETE FROM t WHERE c IN('a','d','f','g','j');
mysql> SELECT * FROM t;
```

```
+------+----+
| c    | i  |
+------+----+
| b    |  2 |
| c    |  3 |
| e    |  5 |
| h    |  8 |
| i    |  9 |
| k    | 11 |
+------+----+
```


下面的 ALTER TABLE 语句将完成先删除再重新创建数据列 i 的工作：

```
mysql> ALTER TABLE t
    -> DROP PRIMARY KEY,
    -> DROP i,
    -> ADD i INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    -> AUTO_INCREMENT = 1;
mysql> SELECT * FROM t;
+------+---+
| c    | i |
+------+---+
| b    | 1 |
| c    | 2 |
| e    | 3 |
| h    | 4 |
| i    | 5 |
| k    | 6 |
+------+---+
```

子句 AUTO_INCREMENT = 1 将使序列从 1 开始重新编号。如果是 MyISAM、MEMORY 或 InnoDB 数据表，你可以用一个不等于 1 的数字使序列从另外一个值开始重新编号。如果是其他的存储引擎，就不必写出 AUTO_INCREMENT 子句了，因为它们不允许这样设定序列编号的初始值，序列将从 1 开始重新编号。

不过，虽然重新编排现有序列编号的工作很容易完成，但往往没有必要这么做。要知道，MySQL 并不在意编号序列里有没有断裂带，而你也不会因为重新编排而在性能方面得到任何好处。此外，如果在另一个数据表里有数据行引用了 AUTO_INCREMENT 数据列里的值，调整该数据列的顺序将破坏数据表之间的对应关系。

3.4.5 如何在不使用AUTO_INCREMENT的情况下生成序列编号

另一种生成序列编号的办法是根本就不使用 AUTO_INCREMENT 数据列，这需要用到带参数的 LAST_INSERT_ID()函数。如果你用函数 LAST_INSERT_ID(*expr*)插入或修改了一个数据列，紧接着又调用了一次不带参数的 LAST_INSERT_ID()函数，那么第二次函数调用所返回的就将是表达式 *expr* 的值。换句话说，MySQL 将把表达式 *expr* 的值视为一个新生成的 AUTO_INCREMENT 值。因此，你可以先生成一个序列编号，然后再在后面的会话中对它进行检索而无须担心这个编号值会受到其他客户程序活动的影响。

这种策略的一种用法是创建一个只有一个数据行的数据表，它里面的值将在你每次需要一个序列编号时被更改一次。比如说，你可以这样创建和初始化这个数据表：

```
CREATE TABLE seq_table (seq INT UNSIGNED NOT NULL);
INSERT INTO seq_table VALUES(0);
```

上面这些语句创建了一个只有一个数据行的数据表 seq_table，其中 seq 的初始值为 0。这个数据表的用法是：先生成一个新的序列编号，再把它检索出来，如下所示：

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq+1);
SELECT LAST_INSERT_ID();
```

上面这条 UPDATE 语句将检索出 seq 数据列的当前值，给它加上 1 以生成序列中的下一个编号。利用函数 LAST_INSERT_ID(seq+1)来生成一个新编号值的做法将使它被 MySQL 视为一个 AUTO_INCREMENT 值，而这个新编号值又可以用不带参数的 LAST_INSERT_ID()函数检索出来。LAST_INSERT_ID()函数是特定于具体客户程序的，即使其他客户程序在你发出 UPDATE 语句和 SELECT 语句的时间间隔里又生成了其他序列编号，你检索到的也仍将是正确值，是你刚生成的那个新编号值。

利用这一策略，我们还能生成出间隔步长不等于 1（甚至是负步长）的序列编号来。比如说，反复执行下面这条语句，我们就能生成出间隔步长等于 100 的序列编号来：

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq+100);
```

而反复执行下面这条语句将生成出按 1 递减（间隔步长等于−1）的序列编号来：

```
UPDATE seq_table SET seq = LAST_INSERT_ID(seq-1);
```

利用这一技巧，只要给 seq 数据列设置一个适当的初始值，就能生成一个从任意值开始编号的序列。

前面的讨论描述了如何利用一个只包含一行的数据表来建立计数器。这在只需要用到一个计数器的场合当然没问题。如果需要用到多个计数器，你可以这样做：给这个表增加一列来充当计数器的标识符，然后在这个表里为每个计数器分别增加另一行。假设你拥有一个网站，打算在几个网页上添加一个“本网页已被访问 *n* 次”的计数器。首先，创建一个有两个数据列的数据表，一列用来保存计数器唯一的名字，另一列用来保存计数器当前值。你仍可以使用 LAST_INSERTED_ID()函数，但要根据计数器的名字来确定用它处理哪一行。比如说，可以用如下所示的语句创建一个这样的数据表：

```
CREATE TABLE counter
(
  name  VARCHAR(255) CHARACTER SET latin1 COLLATE latin1_general_cs NOT NULL,
  value INT UNSIGNED,
  PRIMARY KEY (name)
);
```

name 数据列是一个字符串，你可以给计数器任意命名。同时，为了避免计数器的名字出现重复，把它声明为一个 PRIMARY KEY 前提是，使用表的应用程序知道它将使用的名字。就 Web 计数器来说，要想给它们起一个不重复的名字，需要利用文档树里每个页面的路径名作为计数器名。为了区分路径名里的大小写字母，我们还给 name 数据列加上了区分大小写的排序方式（如果你使用的系统不区分路径名里的字母大小写，就应使用不区分大小写的排序方式）。

在使用 counter 数据表的时候，INSERT ... ON DUPLICATE KEY UPDATE 语句将很有用，它既可以用来为一个未曾统计过的页面插入一个新数据行，也可以用来为一个现有的页面更新其计数值。此外，通过使用 LAST_INSERT_ID(*expr*)函数来生成计数值，你还可以在更新当前计数器

后轻而易举地检索出它的值来。比如说，如果你想对某个网站主页的计数器进行初始化或进行递增，可以这么做：

```
INSERT INTO counter (name, value)
  VALUES('index.html', LAST_INSERT_ID(1))
  ON DUPLICATE KEY UPDATE value = LAST_INSERT_ID(value+1);
SELECT LAST_INSERT_ID();
```

如果你想递增某个现有页面的计数器，但不想使用 LAST_INSERT_ID()函数，可以用另一个办法：

```
UPDATE counter SET value = value+1 WHERE name = 'index.html';
SELECT value FROM counter WHERE name = 'index.html';
```

不过，万一在发出 UPDATE 语句之后、发出 SELECT 语句之前有另外一个客户递增了这个计数器，用这个办法得到的结果将是错误的。可以通过在这两条语句的前后分别加上 LOCK TABLE 和 UNLOCK TABLE 语句来解决这个问题。或者，也可以选用一种支持事务处理的存储引擎来创建这个数据表并使用一个事务来完成这个数据表修改操作。这两种办法都能让你在使用计数器的时候阻断其他客户，但使用 LAST_INSERT_ID()函数显然更容易。因为它的值只与当前客户相关，所以能够确保你得到的是你刚插入的值，绝非来自其他客户，而你在达到目的的同时还用不着使用相对复杂的数据表锁定或事务代码来排斥其他客户。

3.5 表达式求值和类型转换

表达式包含子项和操作符，需要经过求值才能得到结果。子项可以包括诸如常数、函数调用、数据列引用、标量子查询之类的值。这些值可以用不同种类的操作符（例如算术操作符或比较操作符）组合起来，而表达式的子项还可以用括号进行分组。表达式在数据列输出列表和 SELECT 语句的 WHERE 子句里最为常见。比如说，下面是一个查询，它与我们在第 1 章里用来计算年龄的那个查询有不少相似之处。

```
SELECT
  CONCAT(last_name, ', ', first_name),
  TIMESTAMPDIFF(YEAR, birth, death)
FROM president
WHERE
  birth > '1900-1-1' AND DEATH IS NOT NULL;
```

每一个被选取的值都代表着一个表达式，WHERE 子句的内容也是如此。表达式还可以出现在 DELETE 和 UPDATE 语句的 WHERE 子句里、INSERT 语句的 VALUE()子句里，等等。

在遇到表达式时，MySQL 将对它进行求值并产生一个结果。比如说，表达式(4 * 3) DIV (4 - 2)的求值结果是 6。表达式的求值过程往往伴随着各种类型转换操作。比如说，在一个需要 DATE 值的上下文里，MySQL 将把数值 960821 转换为日期值'1996-08-21'。

在这一节里，我们将着重介绍 MySQL 表达式的写法和 MySQL 在对表达式求值的过程中所使用的各种类型转换规则。我们的讨论将涉及 MySQL 的每一种操作符，但论及的函数却只是一小部分，因为 MySQL 的函数实在是太多了。有关 MySQL 函数的详细介绍请参阅附录 C。

3.5.1 表达式的编写

表达式可以简单到只是一个常数，例如数值 0 和字符串'abc'。

表达式里允许出现函数调用。有些函数带有输入参数（即括号里的值），有些则不带。如果某个函数的输入参数多于一个，就要用逗号把它们分隔开。在调用内置函数时，它的输入参数之间允许出现空格。但函数名与紧随其后的左括号之间则不允许出现空格，因为常会出现语法错误。不过，如果你使用了 IGNORE_SPACE，MySQL 就会允许内置函数名的后面出现空格，但它会把函数名全都视为保留字。

表达式里还允许出现数据列引用。在最简单的场合，即 MySQL 能够根据上下文清楚地知道某个数据列属于哪个表时，你可以只给出该数据列的名字。在下面的两条 SELECT 语句里，因为它们都只用到了一个数据表，所以尽管两条语句里的数据列名字完全一样，MySQL 也分得清它们到底属于哪个数据表：

```
SELECT last_name, first_name FROM president;
SELECT last_name, first_name FROM member;
```

如果 MySQL 无法根据数据列的名字把它们区分开来，就必须在数据列名字的前面加上它所在的数据表的名字作为限定。如果 MySQL 无法确定数据表属于哪一个数据库，就必须在数据表名字的前面加上它所在的数据库的名字作为限定。即使在不会导致歧义的场合，你也可以采用这种更为具体明确的形式来书写表达式。如下所示：

```
SELECT
  president.last_name, president.first_name,
  member.last_name, member.first_name
FROM president INNER JOIN member
WHERE president.last_name = member.last_name;

SELECT sampdb.student.name FROM sampdb.student;
```

标量子查询可以在表达式里用来提供单个的值。这样的子查询必须用括号括起来：

```
SELECT * FROM president WHERE birth = (SELECT MAX(birth) FROM president);
```

最后，你可以根据以上这些值（常数、函数调用、数据列引用和子查询）构造出各种复杂的表达式。

1. 操作符的种类

将各种表达式组织在一起的操作符有好几种，本节介绍它们的用途，下一节介绍它们的优先级。

如表 3-18 所示，算术操作符包括常见的加法操作符、减法操作符、乘法操作符、除法操作符和求余操作符。两个操作数都是整数的运算，将使用 BIGINT（64 位）整数值来进行。如果两个操作数都是整数，只要其中一个操作符是无符号的，结果就将是无符号的。对于除 DIV 以外的每个操作符，只要有一个操作数是一个近似值，就会按双精度浮点运算规则进行计算。从字符串转换而来的数值也适用这一原则，因为字符串总是被转换为双精度数值。要特别注意在整数运算中可能出现的超大数值。比如说，如果计算结果超过 64 位，得到的结果将不可预测。（事实上，应该尽量避免使用超过 63 位的整数，因为正负号也要占用一位。）

逻辑操作符（见表 3-19）用来对逻辑表达式的真（非 0）、假（0）进行求值。如果操作数的值无法确定（比如“1 AND NULL”的情况），逻辑表达式还可能被求值为 NULL。

表3-18 算术操作符

操 作 符	语 法	含 义
+	a + b	加法，两个操作数之和
-	a - b	减法，两个操作数之差
-	-a	求负操作符，把操作符取为负值
*	a * b	乘法，两个操作数之积
/	a / b	除法，两个操作数之商
DIV	a DIV b	除法，两个操作数之商的整数部分
%	a % b	求余，除法操作的余数

表3-19 逻辑操作符

操 作 符	语 法	含 义
AND、&&	a AND b、a && b	逻辑与。若两个操作数同时为真，则结果为真
OR、||	a OR b、a || b	逻辑或。只要有一个操作数为真，则结果为真
XOR	a XOR b	逻辑异或。若有且仅有一个操作数为真，则结果为真
NOT、!	NOT a、!a	逻辑非。若操作数为假，则结果为真

作为 AND、OR 和 NOT 操作符的替代品，MySQL 还支持&&、|| 和 ! 操作符，它们的用法和在 C 语言中一样。请特别注意||操作符。SQL 语言标准把||规定为字符串拼接操作符，但它在 MySQL 里完成却的是逻辑“或”操作。如果打算用如下所示的表达式来拼接字符串，就会惊讶地发现其返回结果是数值 0：

```
'abc' || 'def'                                   → 0
```

之所以会发生这样的事情，是因为'abc'和'def'将被转换为整数以进行逻辑“或”操作，而这两个字符串的转换结果都是 0。在 MySQL 里，你必须使用 CONCAT('abc','def')或其他办法来拼接字符串，如下所示：

```
CONCAT('abc','def')                              → 'abcdef'
'abc' 'def'                                      → 'abcdef'
```

如果想让||操作符具有 SQL 语言标准所规定的行为，必须提前启用 PIPES_AS_CONCAT SQL 模式。

表 3-20 中的位操作符用来完成二进制的与、或和异或操作，其结果是依次对两个操作数相应的二进制位进行逻辑“与”、逻辑“或”、逻辑“异或”操作而得到的值。你还可以执行左右移位操作。位操作是使用 BIGINT（64 位）整数值来进行的。

表3-20 位操作符

操 作 符	语 法	含 义
&	a & b	位与。若两操作数的同一位同时为1，则结果中的该位为1
|	a | b	位或。若两操作数的同一位有一个为1，则结果中的该位为1
^	a ^ b	位异或。若两操作数的同一位分别为1和0，则结果中的该位为1
<<	a << b	把a中的各位左移b个位置
>>	a >> b	把a中的各位右移b个位置

比较操作符（见表 3-21）包括用来判断数值和字符串的大小或排位次序的操作符、用来进行模式匹配的操作符以及测试 NULL 值的操作符。<=>操作符是 MySQL 独有的，它最早出现于 MySQL 3.23 版本。

对于字符串比较的属性，参见 3.1.2 节。

表3-21　比较操作符

操　作　符	语　　法	含　　义
=	a = b	若两个操作数相等，则结果为真
<=>	a <=> b	若两个操作数相等（即使为NULL值），则结果为真
<>, !=	a <> b, a != b	若两个操作数不等，则结果为真
<	a < b	若a小于b，则结果为真
<=	a <= b	若a小于或等于b，则结果为真
>=	a >= b	若a大于或等于b，则结果为真
>	a > b	若a大于b，则结果为真
IN	a IN (b1, b2, ...)	若a等于b1、b2……中的某一个，则结果为真
BETWEEN	a BETWEEN b AND c	若a在b和c之间（包括b和c），则结果为真
NOT BETWEEN	a NOT BETWEEN b AND c	若a不在b值和c值之间（并且不等于b或c），则结果为真
LIKE	a LIKE b	SQL模式匹配。若a匹配b，则结果为真
NOT LIKE	a NOT LIKE b	SQL模式匹配。若a不匹配b，则结果为真
REGEXP	a REGEXP b	正则表达式匹配。若a匹配b，则结果为真
NOT REGEXP	a NOT REGEXP b	正则表达式匹配。若a不匹配b，则结果为真
IS NULL	a IS NULL	如果a是NULL值，则结果为真
IS NOT NULL	a IS NOT NULL	如果a不是NULL值，则结果为真

模式匹配不需要你给出精确无误的值就能把数据给检索出来。MySQL 提供了两种模式匹配机制：一种叫 SQL 模式匹配，利用 LIKE 操作符以及通配符“%”（能与任意个字符序列相匹配）和“_”（只能与一个字符相匹配）匹配；另一种是利用 REGEXP 操作符和正则表达式匹配，该机制中的规则表达式与 grep、sed、vi 等 UNIX 程序所使用的正则表达式非常相似。模式匹配只能使用某种模式匹配操作符才能完成；等号操作符（=）只能用来判断两个数据是否相等，不具备模式匹配能力。与模式匹配概念相对立的“模式不匹配”操作可以用 NOT LIKE 或 NOT REGEXP 操作符来完成。

除不同的操作符和不同的模式字符（也就是通配符）外，两种模式匹配机制还在以下方面有着重要的差异。

- ❑ LIKE 操作符可以正确处理多字节数据。REGEXP 操作符只能正确地处理单字节字符集，而且没有把排序方式考虑进来。
- ❑ 只有整个字符串得到匹配时，SQL 模式才算匹配成功；而只要能在字符串里找到匹配，正则表达式模式就算匹配成功。

与 LIKE 操作符配合使用的匹配模式里可以包括通配符“%”和“_”。比如说，模式'Frank%'将匹配任何一个以'Frank'开头的字符串：

```
'Franklin' LIKE 'Frank%'                    → 1
'Frankfurter' LIKE 'Frank%'                 → 1
```

通配符“%”能与任意长度的字符序列（包括空字符序列）匹配，所以模式'Frank%'完全能够与'Frank'匹配：

```
'Frank' LIKE 'Frank%'                              → 1
```

这意味着模式'%'能够与任何一个字符串（包括空字符串）匹配。但'%'不能匹配NULL值。事实上，任何一个模式都不能与NULL值匹配：

```
'Frank' LIKE NULL                                  → NULL
NULL LIKE '%'                                      → NULL
```


只要其操作数中有一个是二进制串，MySQL的LIKE操作符就会把它们都当做二进制串来进行比较。如果两个操作数都不是二进制串，LIKE操作符将根据它们的排序方式对之进行比较，如下所示：

```
'Frankly' LIKE 'Frank%'                            → 1
'frankly' LIKE 'Frank%'                            → 1
BINARY 'Frankly' LIKE 'Frank%'                     → 1
BINARY 'frankly' LIKE 'Frank%'                     → 0
'Frankly' COLLATE latin1_general_cs LIKE 'Frank%'  → 1
'frankly' COLLATE latin1_general_cs LIKE 'Frank%'  → 0
'Frankly' COLLATE latin1_bin LIKE 'Frank%'         → 1
'frankly' COLLATE latin1_bin LIKE 'Frank%'         → 0
```

这种行为与ANSI SQL的LIKE操作符是不一样的，后者是区分字母大小写情况的。

LIKE操作符的另一个通配符是下划线字符“_”，它只能用来匹配单个的字符。比如说，模式'___'将与任何一个有且仅有3个字符的字符串相匹配；而模式'c_t'将与'cat'、'cot'、'cut'，甚至'c_t'（因为'_'能够匹配它自身）相匹配。

通配符可以出现在模式里的任意位置。比如说，模式'%bert'只能与'Englebert'、'Bert'、'Albert'等以'bert'结尾的字符串相匹配；模式'%bert%'除了能与上述字符相匹配外，还能与'Berthold'、'Bertram'、'Alberta'等以'bert'开头或者里面包含'bert'的字符串相匹配；模式'b%t'则能与'Bert'、'bent'、'burnt'等任何一个以字母'b'开头以字母't'结尾的字符串相匹配。

如果想对字符“%”或“_”进行匹配，就必须给它们加上一个前导的反斜线字符（“\%”或“_”）以取消其特殊含义，如下所示：

```
'abc' LIKE 'a%c'                                   → 1
'abc' LIKE 'a\%c'                                  → 0
'a%c' LIKE 'a\%c'                                  → 1
'abc' LIKE 'a_c'                                   → 1
'abc' LIKE 'a\_c'                                  → 0
'a_c' LIKE 'a\_c'                                  → 1
```

MySQL还可以使用正则表达式进行匹配，操作符是REGEXP而不是LIKE。下面是一些比较常用的正则表达式模式字符。

句号字符“.”用来匹配任何一种单个字符：

```
'abc' REGEXP 'a.c'                                 → 1
```

方括号“[...]”用来匹配在方括号内部出现的任何一种字符：

```
'e' REGEXP '[aeiou]'                               → 1
'f' REGEXP '[aeiou]'                               → 0
```

连字符“-”用来给出一个字符范围，只要在字符“-”的两端分别写出这个范围的起始字符和结

束字符就行了。如果在这个字符范围的前面再加上一个“^”字符，就可以用它来匹配那些不属于这一范围的字符了，如下所示：

```
'abc' REGEXP '[a-z]'                    → 1
'abc' REGEXP '[^a-z]'                   → 0
```

星号字符“*”匹配“0个或多个在它之前的那个字符”。比如说，模式'x*'将匹配连续出现的任意个数的'x'字符：

```
'abcdef' REGEXP 'a.*f'                  → 1
'abc' REGEXP '[0-9]*abc'                → 1
'abc' REGEXP '[0-9][0-9]*'              → 0
```

这里所说的“任意次数”也包括0次在内（即允许字符根本没有出现），而这正是上面第二个表达式被求值为1的原因。如果不想包括0（1次及以上），就用“+”代替“*”。

```
'abc' REGEXP 'cd*'                      → 1
'abc' REGEXP 'cd+'                      → 0
'abcd' REGEXP 'cd+'                     → 1
```

'^*pattern*'和'*pattern*$'形式的模板将分别匹配以*pattern*开头或结尾的字符串，而'^*pattern*$'形式的模板将匹配整个字符串是且仅是*pattern*的情况，如下所示：

```
'abc' REGEXP 'b'                        → 1
'abc' REGEXP '^b'                       → 0
'abc' REGEXP 'b$'                       → 0
'abc' REGEXP '^abc$'                    → 1
'abcd' REGEXP '^abc$'                   → 0
```

MySQL的正则表达式模式匹配还有其他一些特殊的模式字符，请参见附录C。

MySQL允许使用数据列的值作为LIKE REGEXP模式，不过，如果该数据列有好几种不同值的话，这种做法要比使用一个常数形式的模式慢——数据列的取值每变化一次，MySQL都不得不重新检查一次有关模式并把它再次转换为内部格式。

2. 操作符的优先级

在对表达式求值时，MySQL会根据操作符的优先级来决定表达式各“零部件”的求值次序。优先级较高的操作符会先于其他操作符得到求值。比如说，乘法和除法的优先级就要比加法和减法的高。请看下面两个例子，它们的求值结果之所以相同，就是因为操作符“*”和“DIV”将在操作符“+”和“-”之前得到求值：

```
3 + 4 * 2 - 10 DIV 2                    → 6
3 + 8 - 5                               → 6
```

我们把操作符按优先级从高到低的顺序依次列出。其中，同一行的操作符的优先级相同，优先级较高的操作符将在优先级较低的操作符之前得到求值，而优先级相同的操作符将按从左至右的顺序依次得到求值。

```
BINARY  COLLATE
!
- (unary minus)  ~ (unary bit negation)
^
*  /  DIV  %  MOD
+  -
<<  >>
```

```
&
|
<  <=  =  <=>  <>  !=  >=  >  IN  IS  LIKE  REGEXP  RLIKE
BETWEEN  CASE  WHEN  THEN  ELSE
NOT
AND  &&
XOR
OR  ||
:=
```

3

根据 SQL 模式或 MySQL 软件版本的不同，有些操作符有不同的优先顺序。参见附录 C。

括号可以用来重写操作符的优先级并改变表达式各“零部件”的求值次序，如下所示：

```
1 + 2 * 3 - 4 / 5                                  → 6.2000
(1 + 2) * (3 - 4) / 5                              → -0.6000
```

3. 表达式中的Null值

要特别注意表达式里出现的 NULL 值，因为其结果往往会出乎你的预料。下面是一些基本的注意事项。

如果把 NULL 值用作算术操作符或位操作符的操作数，其求值结果将是 NULL：

```
1 + NULL                                           → NULL
1 | NULL                                           → NULL
```

如果把 NULL 值用作逻辑操作符的操作数，那么，除非真的有一个确切的结果，否则其求值结果也将是 NULL。

```
1 AND NULL                                         → NULL
1 OR NULL                                          → 1
0 AND NULL                                         → 0
0 OR NULL                                          → NULL
```

如果把 NULL 值用作比较操作符或模式匹配操作符的操作数，那么，除专门用来处理 NULL 值的“<=>”、“IS NULL”和“IS NOT NULL”操作符以外，其他操作符的求值结果都将是 NULL：

```
1 = NULL                                           → NULL
NULL = NULL                                        → NULL
1 <=> NULL                                         → 0
NULL LIKE '%'                                      → NULL
NULL REGEXP '.*'                                   → NULL
NULL <=> NULL                                      → 1
1 IS NULL                                          → 0
NULL IS NULL                                       → 1
```

如果把 NULL 值用作函数的输入参数，那么，除支持 NULL 值作为其输入参数的那些函数以外，其他函数通常都会返回 NULL 值。比如说，IFNULL()就是一个支持 NULL 值作为其输入参数的函数，它将根据输入参数的具体情况返回一个真值或假值。再比如说，STRCMP()函数不支持 NULL 值作为其输入参数，如果你传递给它的输入参数是 NULL 值，它将返回一个 NULL 而不再是一个真值或假值。

在排序操作中，NULL 值都将被集中到一起。它们出现在升序中的最前面，降序中的最后面。

3.5.2 类型转换

只要某个数据值的类型与上下文所要求的类型不相符，MySQL 就会根据将要进行的操作自动地

对该数据值进行类型转换。下面是几种比较常见的需要进行类型转换的场合。

- 根据操作符的求值规则把其他类型的操作数转换为“正确的”类型。
- 根据函数对输入参数的期望把其他类型的输入参数转换为“正确的”类型。
- 根据数据列的类型定义把其他类型的数据值转换为“正确的”类型并插入到该数据列里。

还可以利用一个类型转换操作符或函数来进行显式类型转换。

下面就是一个需要进行隐式类型转换的表达式。它由一个加法操作符（+）和两个操作数（1 和 '2'）构成：

```
1 + '2'                                   → 3
```

既然两个操作数的类型不一致（一个是数值，另一个是字符串），MySQL 就得对它们当中的某一个进行类型转换才能使它们一致。那么，应该对哪一个进行类型转换呢？在这个例子里，“+”是一个数值操作符，它要求两个操作数都是数值类型，所以 MySQL 将先把字符串'2'转换为数值 2，然后再求出表达式的最终结果 3。

我们再来看一个例子。CONCAT()函数能够把多个字符串合并为一个更长的字符串。不管输入参数是什么类型，这个函数都将把它们当做字符串来对待。于是，当你把一些数值传递给 CONCAT()函数时，它会先把它们转换为字符串，然后再把它们合并起来作为返回值，如下所示：

```
CONCAT(1,23,456)                          → '123456'
```

如果 CONCAT()函数又是一个更大的表达式的一部分，就需要经过更多的类型转换才能得到最终的结果。请看下面这个表达式：

```
REPEAT('X',CONCAT(1,2,3)/10)              → 'XXXXXXXXXXXX'
```

这个表达式的求值过程是：首先，CONCAT(1, 2, 3)返回字符串'123'，接着，'123'/10 又被转换为 123/10（因为“/”是一个算术操作符）。表达式 123/10 的结果是 12.3，但因为 REPEAT()函数需要一个整数值来给出字符'X'的重复次数，所以 123/10 在这里进行的是整数除法，计算结果等于 12。最后，REPEAT('X', 12)把字符'X'重复了 12 次，也就是我们看到的字符串结果。

如果 CONCAT()函数的所有参数都是非二进制串，其结果将是一个非二进制串。只要有一个参数是二进制串，其结果就将是一个二进制串。后一条规则包括它有数值型参数的情况，数值会被转换为二进制串。下面这些例子的结果从表面上看是一样的：

```
CONCAT('1','23')                          → '123'
CONCAT(1,'23')                            → '123'
```

但如果你用 CHARSET()函数去检查一下这些结果，就会发现这两个表达式分别返回了一个非二进制串和一个二进制串：

```
CHARSET(CONCAT('1','23'))                 → 'latin1'
CHARSET(CONCAT(1,'23'))                   → 'binary'
```

默认情况下，MySQL 会尽可能地进行类型转换而不是向你报错；这一原则请大家一定要记住。根据上下文，MySQL 会把不符合求值要求的数据类型（数值、字符串、日期/时间）尽可能地转换为它需要的数据类型，但数据并不是总能从一种类型被转换为另一种类型的。如果某给定类型的数据不能被转换为目标类型里的合法数据，那么转换失败。比如说，字符串'abc'根本就不能被转换为一个合法的数值，所以转换结果为 0。同样地，如果某项数据看上去根本不像是个日期/时间值，它在日期/时间上下文里的类型转换结果就将是目标日期/时间类型的“零”值。比如说，把字符串'abc'转换为

一个 DATE 值的结果将是 DATE 类型的“零”值——'0000-00-00'。反之，因为任何数据都可以用字符串来表示，所以把其他类型的数据值转换为字符串通常不会有什么麻烦。

如果想在数据输入操作期间防止把非法值转换为最接近的合法值，可以启用“严格”模式以便系统在发现问题时报告错误。详见 3.3 节。

除上面介绍的以外，MySQL 还能够做一些“小”转换。比如说，如果你在整数上下文里使用了一个浮点数，MySQL 就会按四舍五入的规则把它转换为一个整数。反过来也是如此，整数可以毫无问题地被当做浮点数来使用。

如果上下文没有明确地表明需要的是一个数值，十六进制常数就会被当做二进制字符串来对待。在字符串上下文里，每两个十六进制数字会被转换为一个字符，最终结果将是一个字符串。我们来看一些例子：

```
0x61                                   → 'a'
0x61 + 0                               → 97
X'61'                                  → 'a'
X'61' + 0                              → 97
CONCAT(0x61)                           → 'a'
CONCAT(0x61 + 0)                       → '97'
CONCAT(X'61')                          → 'a'
CONCAT(X'61' + 0)                      → '97'
```

比较操作中的十六进制常数被对待为二进制字符串或数字。

- 这个表达式将把操作数当做二进制串来对待并进行逐字节比较。

```
0x0d0a = '\r\n'                        → 1
```

- 这个表达式把一个十六进制常数和一个数值进行比较，该常数将先被转换为一个数值再进行比较。

```
0x0a = 10                              → 1
```

- 这个表达式进行的是二进制串比较。左操作数的第一个字节的值比右操作数的第一个字节更小，所以结果是“假”。

```
0xee00 > 0xff                          → 0
```

- 在这个表达式里，右操作数中的十六进制常数因为那个算术加法操作符而先被转换为一个数值。为了完成比较操作，左操作数也被转换为一个数值。因为 0xee00（60928）从数值的角度看大于 0xff（255），所以比较操作的结果是“真”。

```
0xee00 > 0xff+0                        → 1
```

可以通过使用一个字符集前导符或 CONVERT() 函数的办法把一个十六进制常数强行转换为一个非二进制串：

```
0x61                                   → 'a'
0x61 = 'A'                             → 0
_latin1 0x61 = 'A'                     → 1
CONVERT(0x61 USING latin1) = 'A'       → 1
```

有些操作符会把自己的操作数强制转换为它期望的类型，而不管那些操作数原来是哪一种类型。算术操作符就是这样做的，它们要求自己的操作数必须是数值，并会强制进行相应的转换：

```
3 + 4                                         → 7
'3' + 4                                       → 7
'3' + '4'                                     → 7
```

在把字符串转换为数字时，并不是让字符串里包含一个数值就足够的。MySQL 不会检查整个字符串以找出其中包含的数值，它只检查字符串的开头。如果字符串的开头部分不是数值，转换结果就将是 0，如下所示：

```
'1973-2-4' + 0                                → 1973
'12:14:01' + 0                                → 12
'23-skidoo' + 0                               → 23
'-23-skidoo' + 0                              → -23
'carbon-14' + 0                               → 0
```

MySQL 的“从字符串转换为数值”的规则适用于把看起来像数值的字符串转换为浮点值：

```
'-428.9' + 0                                  → -428.9
'3E-4' + 0                                    → 0.0003
```

注意，这种转换对看起来像十六进制常数的字符串没有效果。只有前导的“0”还算有点儿用。

```
'0xff' + 0                                    → 0
```

位操作符对操作数的要求比算术操作符还要严格。它们不仅要求操作数是数值，还要求它们必须是整数，并会按照这一要求进行相应的类型转换。这意味着 0.3 这样的浮点数——虽然是一个非零值——不被视为逻辑真值，因为把 0.3 转换为整数所得到的结果将是 0。在下面几个表达式里，小于 1 的操作数都没有被认为是逻辑真值：

```
0.3 | .04                                     → 0
1.3 | .04                                     → 1
0.3 & .04                                     → 0
1.3 & .04                                     → 0
1.3 & 1.04                                    → 1
```

模式匹配操作符用于对字符串进行操作。这意味着对数值也可以使用 MySQL 的模式匹配操作符，因为它在试图匹配时会把它们转换为字符串！

```
12345 LIKE '1%'                               → 1
12345 REGEXP '1.*5'                           → 1
```

数量意义上的比较操作符（<、<=、=等）是上下文相关的，即它们将根据操作数的类型来求值。在下面这个表达式里，两个操作数都是数值，所以它们之间的比较是数值型的：

```
2 < 11                                        → 1
```

在下面这个表达式里，两个操作数都是字符串，所以它们之间的比较是字符串型的：

```
'2' < '11'                                    → 0
```

但在下面两个表达式里，因为两个操作数的类型各不相同，所以 MySQL 将按数值方式对它们比较。因此，两个比较操作的结果都将为真：

```
'2' < 11                                      → 1
2 < '11'                                      → 1
```

在对比较操作求值时，MySQL 将根据以下规则对操作数进行相应的类型转换。

- 除<=>操作符以外，所有涉及 NULL 值的比较操作都将被求值为 NULL。（“<=>”与“=”功能

相当，但可以用来比较 NULL 值。表达式 NULL <=> NULL 将被求值为真，而 NULL=NULL 结果则为 NULL。)

- 如果两个操作数都是字符串值，它们之间的比较操作将按该类型的方式进行。对于二进制字符串，将逐字节地比较各个字节里的数值；对于非二进制字符串，则逐字符地按照它们在字符集里的排序比较。如果两个字符串使用的字符集不同，可能导致错误或不会求出有意义的结果。如果两个操作数一个是普通的字符串，另一个是二进制字符串，它们之间的比较操作就将按二进制字符串方式进行。
- 如果两个操作数都是整数，它们之间的比较操作将按整数方式进行。
- 只要十六进制常数不是与数值进行比较，就都将被视为二进制字符串。
- 除了 IN()，如果比较操作的两个操作数一个是 TIMESTAMP 或 DATETIME 值，另一个是一个常数，比较操作就会把它们都看成 TIMESTAMP 值，这是为了使比较操作与各种 ODBC 应用程序更好地配合。
- 除上述各种情况外，比较操作中的操作数都将被视为双精度浮点数。注意，字符串与数值之间的比较操作就属于这种情况。字符串将被转换为一个双精度数字，如果字符串看起来不像数字，转换结果就将是 0。比如说，字符串'14.3'将被转换为浮点数 14.3，但字符串'L4.3'却会被转换数值 0。

1. 日期/时间值的解释规则

根据表达式中上下文的要求，MySQL 会把字符串和数值自动转换为日期/时间值（或相反）。在数值上下文里，日期/时间值将被自动转换为数值。在日期/时间上下文里，数值将被自动转换为日期/时间值，当你对某个日期/时间数据列赋值或使用了一个需要以日期/时间值为输入参数的函数时，会进行这种转换。在比较操作中，一般规则是把日期/时间值当做一个字符串来比较。

比如说，假设数据表 mytbl 里有一个名为 data_col 的 DATE 数据列，那么下面那几条语句将是等价的：

```
INSERT INTO mytbl SET date_col = '2025-04-13';
INSERT INTO mytbl SET date_col = '20250413';
INSERT INTO mytbl SET date_col = 20250413;
```

在下面的例子里，TO_DAYS()函数的输入参数分别是 3 种不同类型的表达式，但它们都将被解释为相同的值：

```
TO_DAYS('2025-04-13')                    → 739719
TO_DAYS('20250413')                      → 739719
TO_DAYS(20250413)                        → 739719
```

2. 预查类型转换结果与强制进行类型转换

如果想预知在某个表达式的求值过程中都会发生什么样的类型转换，可以在 mysql 客户程序里提前用一条 SELECT 语句对这个表达式求值：

```
mysql> SELECT X'41', X'41' + 0;
+-------+-----------+
| X'41' | X'41' + 0 |
+-------+-----------+
| A     |        65 |
+-------+-----------+
```

如果无法观察出某个表达式的类型，可以把它放到一个新的数据表里查看新数据表的定义：

```
mysql> CREATE TABLE t SELECT X'41' AS col1, X'41' + 0 AS col2;
mysql> DESCRIBE t;
+-------+--------------+------+-----+---------+-------+
| Field | Type         | Null | Key | Default | Extra |
+-------+--------------+------+-----+---------+-------+
| col1  | varbinary(1) | NO   |     |         |       |
| col2  | double(17,0) | NO   |     | 0       |       |
+-------+--------------+------+-----+---------+-------+
```

对于那些用来删改数据行的 DELETE 或 UPDATE 语句，提前预查表达式求值结果的做法有着重要的意义，因为你必须确保这些语句将只施加在你想对之进行删改的数据行上。你应该这样做：在执行 DELETE 或 UPDATE 语句之前，先用一条 SELECT 语句对 WHERE 子句里的表达式做一下检查，看它选取出来的数据行是不是你想要的。比如说，假设你的 mytabl 数据表里有一个名为 char_col 的 CHAR 数据列，里面存放着以下数据：

```
'abc'
'00'
'def'
'00'
'ghi'
```

那么，下面这条语句会把哪些东西删除掉呢？

```
DELETE FROM mytbl WHERE char_col = 00;
```

你的本意可能只是想把包含'00'值的那两个数据行删除掉，可这条 DELETE 语句的实际执行效果却是把所有的数据行全都给删除掉了——既让你大吃一惊，又让你后悔莫及！为什么会出现这样的结果呢？这全是 MySQL 的类型转换规则惹的祸。char_col 是一个字符串数据列，但上面这条 DELETE 语句里的 00 却因为没有放在引号里而被当做是一个数值。根据 MySQL 的类型转换规则，字符串与数值进行比较时都被当做数值。因此，在 DELETE 语句的执行过程中，char_col 数据列里的每个值将先转换为一个数值，然后再与数值 0 比较。灾难就这样发生了：'00'被理所当然地转换成了 0，可其他字符串——因为看起来都不像数值——也无一幸免地被转换成了 0。于是，WHERE 子句在每一个数据行上都成立，而 DELETE 语句也就顺理成章地把整个数据表删除得干干净净。要是你在执行 DELETE 语句之前先用一条 SELECT 语句对 WHERE 子句里的表达式进行了检查，就肯定会注意到它选取出来的数据行多得超出了你的预想，你也就有机会避免刚才的悲剧了：

```
mysql> SELECT char_col FROM mytbl WHERE char_col = 00;
+----------+
| char_col |
+----------+
| abc      |
| 00       |
| def      |
| 00       |
| ghi      |
+----------+
```

在拿不准应该如何使用某项数据时，就应该利用 MySQL 的类型转换机制或类型转换函数把它强行转换为你需要的类型。

给数据加上+0 或者加上+0.0 将把它强行转换为一个数值类型的值：

```
0x65                                          → 'e'
0x65 + 0                                      → 101
0x65 + 0.0                                    → 101.0
```

如果想舍弃某个数值的小数部分，可以使用 FLOOR()或 CAST()函数。如果想给某个整数增加一个小数部分，给它加上一个诸如 0.0、0.00 形式的小数（小数点后面的 0 的个数表明你想让该整数具有的精确度）即可：

```
FLOOR(13.3)                                   → 13
CAST(13.3 AS SIGNED)                          → 13
13 + 0.0                                      → 13.0
13 + 0.0000                                   → 13.0000
```

如果想得到四舍五入的效果，就要用 ROUND()函数来代替 FLOOR()函数。

CAST()或 CONCAT()函数能够把任何类型的值转换为字符串：

```
14                                            → 14
CAST(14 AS CHAR)                              → '14'
CONCAT(14)                                    → '14'
```

如果必须把一个数值型参数转换为字符串形式，CONCAT()函数将返回一个二进制串，所以最后两个例子的结果其实是不同的。CAST()表达式返回的是一个非二进制串，而 CONCAT()表达式返回的是一个二进制串。

HEX()函数可以把一个数值转换为一个十六进制串：

可以用 HEX()函数把一个数字值转换为一个十六进制数字串：

```
HEX(255)                                      → 'FF'
HEX(65535)                                    → 'FFFF'
```

HEX()函数还可以把字符串转换为十六进制数字串，字符串里的每个字符将依次被转换两个十六进制数字：

```
HEX('abcd');                                  → '61626364'
```

ASCII()函数能够把一个单字节字符转换为它的 ASCII 编码值：

```
'A'                                           → 'A'
ASCII('A')                                    → 65
```

如果想把 ASCII 编码反向转换为字符，就需要使用 CHAR()函数：

```
CHAR(65)                                      → 'A'
```

DATE_ADD()或 INTERVAL 函数能够把字符串或者数值转换为日期值：

```
DATE_ADD(20080101, INTERVAL 0 DAY)            → '2008-01-01'
20080101 + INTERVAL 0 DAY                     → '2008-01-01'
DATE_ADD('20080101', INTERVAL 0 DAY)          → '2008-01-01'
'20080101' + INTERVAL 0 DAY                   → '2008-01-01'
```

如果给日期值加上一个 0，就可以把它强行转换为数值：

```
CURDATE()                                     → '2007-09-07'
CURDATE()+0                                   → 20070907
```

包括时、分、秒的日期/时间值将被转换为一个带有微秒部分的值：

```
NOW()                                         → '2007-09-07 16:15:29'
NOW()+0                                       → 20070907161529.000000
```

```
CURTIME()                                    → '16:15:29'
CURTIME()+0                                  → 161529.000000
```

为了舍弃小数部分，像下面这样把值转换为一个整数即可：

```
CAST(NOW() AS UNSIGNED)                      → 20070907161529
CAST(CURTIME() AS UNSIGNED)                  → 161529
```

你可以用 COVERT()函数把字符串从一个字符集转换到另一个字符集。使用 CHARSET()函数可以检查结果是否是想要的字符集：

```
'abcd'                                       → 'abcd'
CONVERT('abcd' USING ucs2)                   → 'abcd'
CHARSET('abcd')                              → 'latin1'
CHARSET(CONVERT('abcd' USING ucs2))          → 'ucs2'
```

在一个字符串的前面放上一个字符集前导符并不会导致该字符串被转换，但 MySQL 在解释该字符串时将按照那个前导符的指示来对待它：

```
CHARSET(_ucs2 'abcd')                        → 'ucs2'
```

为了确定某个特定的十六进制 UCS-2 字符所对应的 UTF-8 字符的十六进制值，可以把 CONVERT()和 HEX()函数结合起来。下面的表达式给出了注册商标符号的 UTF-8 值：

```
HEX(CONVERT(_ucs2 0x2122 USING utf8))        → 'E284A2'
```

若想改变字符串的排序方式，可以使用 COLLATE 操作符。若想检查一下结果排序方式是不是你想要的，可以使用 COLLATE()函数：

```
COLLATION('abcd')                            → 'latin1_swedish_ci'
COLLATION('abcd' COLLATE latin1_bin)         → 'latin1_bin'
```

字符集和排序方式必须是兼容的。如果它们不兼容，可以先用 CONVERT()函数转换字符集、再用 COLLATE 操作符改变排序方式：

```
CONVERT('abcd' USING latin2) COLLATE latin2_bin
```

若想把一个二进制串转换为一个有着给定字符集的非二进制串，可以使用 CONVERT()函数：

```
0x61626364                                   → 'abcd'
0x61626364 = 'ABCD'                          → 0
CONVERT(0x61626364 USING latin1) = 'ABCD'    → 1
```

还有，对于十六进制串或用引号括起来的二进制串，可以用一个字符集前导符来改变 MySQL 对它的解释：

```
_latin1 0x61626364 = 'ABCD'                  → 1
```

BINARY 关键字可以把一个非二进制串强制转换为一个二进制串：

```
'abcd' = 'ABCD'                              → 1
BINARY 'abcd' = 'ABCD'                       → 0
'abcd' = BINARY 'ABCD'                       → 0
```

3.6　数据类型的选用

3.2 节描述了各种可供选用的数据类型和那些数据类型的基本特性，如它们所能容纳的值的类型，

它们会占用多少存储空间，等等。但在创建数据表时又该如何决定使用哪种类型呢？本节将要讨论的问题可以帮助大家做出最好的选择。

字符串类型是最“通用”的数据类型。可以把任何东西保存在它们里面，因为数值和日期可以用字符串形式来表示。那么，是不是应该把所有的数据列都定义为字符串并满足于此呢？不是的。我们来看一个简单的例子。假设有一些看起来像是数值的数据。你当然可以把它们表示为字符串，但真的应该这么做吗？如果这么做了，会发生什么事情？

3

首先，你可能会耗用更多的空间，因为使用数值类型的数据列来保存数值要比使用字符串类型更有效率。你还将注意到因为对数值和字符串的处理方式的不同而表现在查询结果方面的一些差异。比如说，数值的排序结果与字符串的是不一样的。数值 2 小于数值 11，但字符串'2'在词法上却是大于字符串'11'的。这个问题可以通过把数据列放到一个数值上下文环境里使用的办法来解决，如下所示：

```
SELECT col_name + 0 as num ... ORDER BY num;
```

给数据列加上 0 就可以强行按数值方式进行排序，但这么做合理吗？这在某些场合里是一个很有用的技巧，但犯不上每次需要按数值方式排序时都用这一招吧。让 MySQL 把一个字符串数据列当做一个数值数据列来对待有几个很重要的问题。首先，这将迫使 MySQL 对数据列里的每一项数据进行从字符串到数值的转换，而这种做法会降低效率。其次，使用这样的数据列进行计算会导致 MySQL 不使用这些数据列上的任何索引，这将进一步降低查询的速度。如果从一开始就把数据保存为数值，就不会发生这两种会导致性能降低的事情了。

刚才的例子揭示了我们在挑选数据类型时需要考虑的几个因素。未经深入分析就选用一种表示形式取代另一种表示形式的简单做法往往会在存储要求、查询处理和整体性能方面导致不良后果。下面列出了一些我们在为数据列挑选类型时应该思考的问题。

这个数据列将容纳什么样的数据？数值？字符串？日期？坐标值？这个问题并不难回答，关键在于你必须向自己提出这个问题。你完全可以把任何类型的数据都表示为字符串，但正如我们刚才看到的那样，如果你能为数值形式的数据挑选一种更适用的类型的话，你得到的回报将是更好的性能。（对日期/时间和坐标形式的数据来说也是如此。）请注意，对你将要处理的数据的类型进行评估并不见得总是那么轻而易举，尤其是在数据来源不是你本人的时候。如果你正在为别人设计一个数据表，把“这个数据列将容纳什么样的数据”这个问题弄清楚将非常重要。要想做出一个最佳的决策，你必须保证你已经问过足够多的问题、收集到了足够多的信息。

数据是否都在某个特定的区间内？如果它们是整数，是不是总是非负值？如果是，你可以考虑选用 UNSIGNED。如果它们是字符串，它们是不是总来自一个固定的、有限的集合？如果是，你将发现 ENUM 或 SET 是一种很有用的类型。

数据类型的取值范围和空间占用量是相互影响的。需要一种多“大”的类型？对于数值，可以选择一种取值范围有限的小类型，也可以选择一种更大的类型。对于字符串，应该根据它们的长短来做出选择。如果打算存储的数据没有超过 10 个字符的，就用不着选择 CHAR(255)。

在性能和效率方面有没有需要考虑的因素？有些类型可以比其他类型处理得更有效率。数值操作通常都比字符串操作执行得更快。短字符串的比较操作要比长字符串的完成得更快速，硬盘方面的开销也更小。就 MyISAM 数据表而言，长度固定的数据行会比长度可变的数据行有更好的性能表现。

以下各节将对这些问题做更详细的讨论，尤其是性能问题，我们将在 5.3 节进行探讨。

在继续前进之前，我向提醒大家一句：虽然你已经在创建数据表时尽最大努力做出了最好的数据

类型选择，但万一你的选择被事实证明不是最优的，也不会面临世界末日。可以利用 ALTER TABLE 语句把不妥当的类型改成一种更好的。事情也许很简单：如果发现某个整数数据列里的数据比你当初预想的大，把它从 SMALLINT 改成 MEDIUMINT 够不够？事情也许会比较复杂，比如把一个 CHAR 数据列改变为一个 ENUM 来容纳它的可取值。可以利用 PROCEDURE ANALYSE()来获得关于数据表里的数据列的信息，如最大值、最小值、MySQL 根据数据列的取值范围而建议使用的优化类型，等等。

```
SELECT * FROM tbl_name PROCEDURE ANALYSE();
```

这个查询的输出报告可以帮助判断是否可以选用一种更“小”的类型，这可以改善相关数据表的查询性能并减少数据表的存储空间占用量。关于 PROCEDURE ANALYSE()更详细的信息请参见 5.3 节。

3.6.1　数据列将容纳什么样的数据

在挑选数据类型时，首先要考虑的事情应该是这个数据列将用来容纳什么样的数据，因为这对你选择的类型有着最直接的影响。一般来说，应该做简明的决定：把数值保存在数值型数据列里，把字符串保存在字符串型数据列里，把日期和时间保存在日期/时间型数据列里。如果数值有一个小数部分，就应该选用 DECIMAL 或浮点类型，而不是某种整数类型。但是例外情况总是难免的。这里的原则是，必须了解你的数据才能胸有成竹地选出最适用的类型。如果数据来自你本人，你也许早就对如何取舍心中有数。从另一方面讲，如果是别人让你替他们设计一个数据表，事情就不一样了，你想了解的东西恐怕没那么容易搞清楚。一定要问足够多的问题，只有这样，才能发现那个数据表到底是用来容纳什么类型的数据的。

假设别人让你帮忙设计一个数据表，其中有一个数据列是用来记录“降雨量”的。它们是数值吗？或者“几乎”（通常是，但并非总是）是数值？比如说，在看电视新闻的时候，天气预报员经常会说到降雨量。有时候，那是一个数值（比如“降雨量 0.25 英寸”），但有时只是说降雨量“稀少”，意思是“几乎没有”。这在天气预报节目里很正常，但怎样才能把这个“稀少”存储到数据库里呢？办法之一是把“稀少”定义为一个数值，以便选用一种数值类型来记录降雨量；办法之二是选用一种字符串类型，以便直接记录“稀少”这两个字。当然，你还可以想出一些更复杂的办法。比如说，同时使用一个数值数据列和一个字符串数据列来记录降雨量，视情况填写其中一个数据列，另一个则为 NULL。不过，只要有可能，你肯定不会选择最后这个办法，那不仅会让数据表变得难以理解，还会大大增加编写查询代码的难度。

如果让我来设计这个数据表，我可能会尽量把所有的数据行都存储为数值形式，然后在必要时对它们进行转换以便于显示。我们不妨假设小于 0.01 英寸且不等于零的降雨量都可以归类为“稀少”，我们就可以用下面这样的代码来显示这个数据列里的值：

```
SELECT IF(precip>0 AND precip<.01,'trace',precip) FROM ... ;
```

有些数据明显是数值，但你必须判断应该选用一种整数类型还是一种浮点类型。你必须把数据的计量单位和精确度弄清楚。整数级的计量单位能否满足精确度要求？是否需要增加一个小数部分？这个问题可以帮你区分整数、定点或浮点数据类型。比如说，如果你记录的“重量”只要求精确到千克，就可以选用一个整数数据列。如果记录要求精确到某位小数，就应该选用一个定点或浮点数据列。在某些特殊场合，你甚至可以使用多个数据列。比如说，分别以“千克”和“盎司”为单位来记录重量。

高度是一种可以用多种办法来表示的数值型信息。

- 用'6-2'这样的字符串来表示“6 英尺，2 英寸”。这种表示形式的优点是易于阅读和理解（肯

定比“74 英寸”好得多)，但这样的值难以用来进行加法或求平均值之类的算术操作。

- 用一个数值数据列来记录英尺数，用另一个来记录英寸数。这可以让算术操作变得稍微容易点儿，但使用两个数据列怎么说也比只使用一个数据列时的情况来得复杂些。
- 用一个数值数据列来表示英寸数。这对数据库来说是最容易处理的，对人类来说却是最不利的。但不要忘了，数据的显示格式和它们的存储/处理格式并非必须保持一致。你可以利用 MySQL 中的许多函数重新编排数据的格式，把它们以易于阅读和理解的形式呈现给人们。那意味着这可能是表示“高度”的最佳办法。

货币（例如美元），是另一种数值型信息。在涉及金钱的计算里，人们需要和由美元和美分构成的数据打交道。它们乍看上去应该是些浮点值，但 FLOAT 和 DOUBLE 类型难免出现四舍五入错误，只适合用来处理对精确度要求不高的数据项。因为人们对自己的金钱都很敏感，所以你肯定需要一种可以提供完美精确度的数据类型。你有以下几种选择。

- 你可以把金额表示为一种 DECIMAL(*M*,2)类型，其中 *M* 是你需要的取值范围的最大宽度。这种类型的数值精确到小数点后面两位。DECIMAL 类型的优点是：数据值不存在四舍五入的错误，计算很精确。
- 你可以在系统内部选用一种整数类型以“分”为单位来表示所有与金钱有关的数据。这种表示方法的优点是计算过程在系统内部是使用整数进行的，速度快；缺点是在输入或输出的过程中必须通过对数据乘以或者除以 100 的办法来完成必要的转换。

某些“数字”其实根本不是真正的数值。电话号码、信用卡号码和社会保险号码都包含有不是数字的字符（比如空格和短划线），不能直接存储到一个数值类型的数据列里，除非你把其中的非数字字符都去掉。但即使把其中的非数字字符都去掉了，你恐怕还是更愿意把它们存储为字符串而不是数字，这样可以避免漏掉开头部分的“零”。

如果你需要保存一些日期信息，你应该问自己：它们包含一个时间值吗？或者，它们是否真的需要包含一个时间值？MySQL 没有提供一种时间部分可选的日期类型：DATE 没有时间部分，DATETIME 必须有时间部分。如果时间部分确实是可选的，那就用一个 DATE 数据列来记录日期，再用一个 TIME 数据列来记录时间好了。然后允许那个 TIME 数据列可以取值为 NULL，并把它解释为“无时间”：

```
CREATE TABLE mytbl
(
  date DATE NOT NULL,     # date is required
  time TIME NULL          # time is optional (may be NULL)
);
```

在某些场合，例如你需要为两个数据表建立“主从”关系而它们之间的“纽带”是日期信息的时候，判断是否需要一个时间值往往非常关键。假设你的研究需要进行一系列测试。在标准的主测试之后，你可能还需要进行几项辅助测试，具体进行哪几项辅助测试要根据主测试的结果来决定。你可以用一个“主从”关系来表示这些信息：把测试主题和标准主测试的情况保存在一个主控数据表里，把辅助测试的情况另行保存在一个细节数据表里，然后通过测试主题 ID 和测试日期把这两个数据表关联在一起。

在这个场景里，必须回答的问题是：只有日期够不够？日期和时间是不是都必不可少？这取决于你是否会在同一天多次进行同一项测试。如果是，就需要记录一个时间值（比如说，测试的开始时间），这既可以单独使用一个 DATETIME 数据列来实现，也可以使用一个 DATE 数据列和一个 TIME 数据列，这两个数据列都是必须填写的。如果没有记录时间值，一旦你在一天之内进行了两次某项测试，你将

无法把该项测试的细节数据行和相应的主控数据行正确地关联在一起。

我曾经听过某些人这么说："我不需要记录一个时间值，我从不在同一天做两次同样的测试。"有时候，他们还真的能说到做到。但我也见过不少这样的人后来焦头烂额地设法防止细节数据行与错误的主控数据行关联在一起，因为他们在一天之内做了多次同样的测试并在输入测试数据之后发现事情变得一团糟。很遗憾，到那时已经太晚了！

有时候，你可以通过往数据表里增加一个 TIME 数据列来解决这个问题。但如果你没有保留原始数据来源（比如原始书面记录）的话，已经输入到数据表里的数据行往往很难整理，你将根本无法把同一天进行的多次测试的细节数据行分别与其正确的主控数据行关联在一起。即使你保留了一份原始数据来源，这种整理也会非常麻烦，而且很可能会导致你当初为使用这些数据表而编写的应用程序出问题。总而言之，最好的解决办法是把这类问题向数据表的拥有者解释清楚，并确保自己在开始创建数据表之前已经对有关数据的各项特征了然于心。

有时候，你一开始只掌握了一些不够完整的数据，而这将影响到你对数据类型的选择。比如说，假设你正在收集某些人的出生和死亡日期以进行家谱研究，而你能收集到的资料只包括那些人的出生或死亡年份或者是年份和月份，没有准确的日子。如果选用一个 DATE 数据列来记录这些信息，你收集到的数据将无法输入——它们都是不完整的日期值，除非你能补足"日子"部分。要想在这种信息不够完整的情况下仍能把已经收集到的数据记录下来，可供选择的最佳方案大概是创建 3 个数据列来分别记录年、月和日，然后把已经收集到的信息输入相应的数据列，剩下的则填入 NULL。另一个办法是坚持使用 DATE 值，如果日期值不够完整，就把相应的日或者月和日设置为 0。这种"模糊"的日期可以用来表示不完整的日期值。

3.6.2　数据是否都在某个特定的区间内

在为某个数据列挑选数据类型时，把基本类型（整数、浮点数、字符串）确定下来之后，思考一下取值范围将有助于把选择面逐步缩窄到该基本类型中的某个特定类型上。假设需要存储一些整数值，你应该根据它们的取值范围来决定选用哪一种具体的整数类型。如果它们的取值范围是 0 到 1000，可供选择的整数类型将是从 SMALLINT 到 BIGINT。如果它们的取值上限是 200 万，SMALLINT 就不够用了，可供选择的整数类型将只能是从 MEDIUMINT 到 BIGINT。

当然，你完全可以只挑选一种最"大"的类型来容纳你想保存的数据（例如为上段中的例子选择 BIGINT）。但一般应该选择足以满足你使用目的的最小类型。这将使数据表占用的存储空间最小化，随之而来的是更好的性能，这是因为比较短小的数据列往往比那些比较长大的数据列处理得更快。（读取比较短小的数据值所需要的磁盘读写次数更少，让键字缓存容纳更多的键字，加快索引搜索的速度。）

如果无法提前获知自己将要处理的数据的取值范围，就只能靠猜测或是选择 BIGINT 之类的"大"类型来预防最坏的可能性。如果靠猜测而选用的类型偏小了，事情还有补救：用 ALTER TABLE 语句及时"加大"的数据列。

有时候，你甚至会发现可以把某个数据列设置得更小。在第 1 章里，我们为考试成绩项目创建了一个 score 数据表，它有一个 score 数据列来记录学生们的考试和测验成绩。为方便当时的讨论，我们在创建该数据列时使用的是 INT 类型。根据上面的讨论，如果学生考试成绩的取值范围是 0 到 100 的话，TINYINT UNSIGNED 将是一个更好的选择，因为它的存储空间占用量更少。

数据的取值范围还会影响到你可以为你选用的数据类型搭配使用哪些属性。比如说，如果数据不

可能是负数，就可以加上 UNSIGNED 属性；反之，则不可以。

字符串类型没有像数值型数据列那样的“取值范围”，但它们都有一个长度范围，你所需要的最大长度对你可以选用哪些数据列类型有着决定性影响。如果需要存储的字符串都比 256 个字符短，可以选用 CHAR、VARCHAR 或 TINYTEXT。如果需要存储更长的字符串，可以选用 VARCHAR 或某种更长的 TEXT 类型。

如果某个数据列所容纳的字符串来自一个有限集合，就应该考虑使用 ENUM 或 SET 数据类型。它们可能是很好的选择，因为它们在系统内部被表示为数值，对它们的操作以数值方式进行，所以它们的处理效率比其他字符串类型更高。它们往往比其他字符串类型更紧凑，因而更节约空间。还有，如果启用了“严格”SQL 模式，还可以防止那些没在合法值清单里的字符串进入数据库。详见 3.3 节。

在确定你将与之打交道的数据的取值范围时，“总是”和“绝不”是两个最有用的词（例如“总是小于 1 000”或“绝不是负数”），它们可以帮你更准确地选出最适用的数据类型。但千万注意不要把这两个词用在不恰当的地方。尤其是在你向别人了解数据情况时，如果对方说出了这两个词，就更要注意。在人们说“总是”或“绝不”时，一定要确认他们的话能不能当真。有时候，当人们说他们的数据“总是”具备某种特征时，实际情况却是“差不多总是”。

假设你正在为一些教师设计一个数据表，他们告诉你：“学生们的考试分数总是在 0 到 100 之间。”因为这句话，你决定选用 TINYINT 类型，又因为数据都是非负数，所以你决定再加上 UNSIGNED 限定符。可你后来发现这些人在把数据录入数据库时有时会用“-1”来表示“学生因病缺考”。糟糕，他们当初可没告诉你还有这个！用 NULL 来代表这类特殊情况是个可以考虑的补救办法，但万一对方不认可，你将不得不记录一个“-1”，而这意味着你不能使用 UNSIGNED 数据列。（在遇到这种麻烦的时候，ALTER TABLE 语句可以帮上大忙。）

有时候，多问一个简单的问题就可以轻松消除许多未来的烦恼：会不会有例外？只要存在某种例外，哪怕只有一例，都必须把它考虑进来。在与别人探讨数据库的设计问题时，你会发现有许多人有这样一种思维误区：只要例外情况不是经常发生，它们就无足轻重。在设计数据表时候可千万不能这么想。你需要提出的问题不是“例外经常发生吗？”而是“有没有发生过例外？” 只要发生过例外，就必须把它们考虑进来。

3.6.3 与挑选数据类型有关的问题是相互影响的

千万不要以为与挑选数据类型有关的问题是彼此无关的。以数值类型为例，其取值范围与其存储空间占用量有关：取值范围越大，需要的存储空间就越大，而这又会影响到性能。创建一个 AUTO_INCREMENT 数据列来容纳唯一化的序列编号也有许多问题需要提前考虑到。这只是个简单的选择，但对数据类型、索引和 NULL 值的使用情况都会产生影响。

- AUTO_INCREMENT 是一种数据列属性，最适合用于整数类型。这立刻就把我们的选择余地限制在了从 TINYINT 到 BIGINT 的范围内。
- AUTO_INCREMENT 数据列只能用来生成一个正整数序列，所以应该把它定义为 UNSIGNED。
- AUTO_INCREMENT 数据列必须有索引。不仅如此，为防止该数据列里出现重复的值，它的索引还必须是唯一的，所以应该把该数据列定义为一个 PRIMARY KEY 或者是一个 UNIQUE 索引。
- AUTO_INCREMENT 数据列必须具备 NOT NULL 属性。（即使你省略了 NOT NULL，MySQL 也会自动地加上它。）

以上这些意味着不能像下面这样定义一个 AUTO_INCREMENT 数据列：

```
mycol arbitrary_type AUTO_INCREMENT
```

必须像下面这样来定义它：

```
mycol integer_type UNSIGNED NOT NULL AUTO_INCREMENT,
PRIMARY KEY (mycol)
```

或者像下面这样来定义它：

```
mycol integer_type UNSIGNED NOT NULL AUTO_INCREMENT,
UNIQUE (mycol)
```

第4章 存储程序

4

MySQL 支持把几种对象存放在服务器端供以后使用。这几种对象有一些可以根据情况通过程序代码调用，有一些会在数据表被修改时自动执行，还有一些可以在预定时刻自动执行。它们可以分为以下几种。

- *存储函数*（stored function）。返回一个计算结果，该结果可以用在表达式里。
- *存储过程*（stored procedure）。不直接返回一个结果，但可以用来完成一般的运算或是生成一个结果集并传递回客户。
- *触发器*（trigger）。与数据表相关联，当那个数据表被 INSERT、DELETE 或 UPDATE 语句修改时，触发器将自动执行。
- *事件*（event）。根据时间表在预定时刻自动执行。

MySQL 对存储函数和存储过程的支持始于 5.0.0 版本，对触发器和事件的支持分别始于 5.0.2 版本和 5.1.6 版本。这些对象类型的早期实现多少会有一些不完善的地方，所以如果你使用的是 MySQL 5.0 或 5.1 系列的版本，最好把它升级到最新的版本以避免那些不足。

存储程序有以下优点和能力。

- 存储程序对象的可执行部分可以用复合语句来编写，复合语句对 SQL 语法进行了扩展，可以包括代码块、循环和条件语句。（所有这些语句的语法可以在 E.1 节里查到。）
- 存储程序都被保存在服务器端，定义它们所需要的代码只需在它们被创建时通过网络传递一次，而不是每次执行都要传递一次。这大大减少了开销。
- 它们可以把复杂的计算封装为程序单元，而你可以简单地通过程序单元的名字来调用它们。
- 它们可以用来实现“标准化”的计算操作。你可以把一组存储程序打包为一个“函数库”供其他应用程序调用，让那些应用程序以同样的方式完成操作。
- 它们提供了一种错误处理机制。
- 它们可以提高数据库的安全性。你可以通过选择存储程序执行时所需的权限下来对敏感数据的访问情况进行限制和调控。

本章使用了以下术语。

- *存储程序*。泛指各种类型的存储对象（存储函数、存储过程、触发器、事件）。
- *存储例程*（stored routine）。特指存储函数和存储过程。这两种对象的定义语法很相似，所以很自然地把它们放在一起讨论。事实上，“存储过程”这个术语常用来统称存储过程和存储函数。但我个人认为那容易引起一些误会，所以我在这本书里没有那么做。

在本章后面的小节里，我们将讨论如何编写和使用每一种存储程序。不过，在开始讨论各种特殊

类型的存储程序的细节之前，我们将先讨论一个和它们都有关的问题：如何编写复合语句。

4.1　复合语句和语句分隔符

简单的存储程序只包含一条 SQL 语句，在编写时不需要特殊对待。下面的存储过程使用了一条 SELECT 语句来列出 sampdb 数据库里的数据表的名字：

```
CREATE PROCEDURE sampdb_tables ()
  SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
  WHERE TABLE_SCHEMA = 'sampdb' ORDER BY TABLE_NAME;
```

不过，存储程序并非只能包含一条简单的 SQL 语句。它们可以包含多条 SQL 语句，可以使用局部变量、条件语句、循环和嵌套语句块等多种语法构造。要使用这些构造编写存储程序，就需要用到复合语句。复合语句由 BEGIN 开头，END 结束，在它们之间可以写出任意数量的语句，这些语句构成了一个语句块。下面的存储过程将显示一条欢迎消息，其中有你的用户名；如果你是一位匿名用户，用户名将是“earthing”：

```
CREATE PROCEDURE greetings ()
BEGIN
  # 77 = 16 for username + 60 for hostname + 1 for '@'
  DECLARE user CHAR(77) CHARACTER SET utf8;
  SET user = (SELECT CURRENT_USER());
  IF INSTR(user,'@') > 0 THEN
    SET user = SUBSTRING_INDEX(user,'@',1);
  END IF;
  IF user = '' THEN         # anonymous user
    SET user = 'earthling';
  END IF;
  SELECT CONCAT('Greetings, ',user, '!') AS greeting;
END;
```

在使用复合语句时，必须考虑和解决这样一个问题：复合语句块里的语句必须以分号（;）彼此隔开，但因为分号同时也是 mysql 程序默认使用的语句分隔符，所以在使用 mysql 程序定义存储程序时会发生冲突。解决这个问题的办法是使用 delimiter 命令把 mysql 程序的语句分隔符重定义为另一个字符或字符串，它必须是在存储例程的定义里没有出现过的。这样一来，mysql 程序就不会把分号解释为语句终止符了，它将把整个对象定义作为一条语句传递给服务器。在定义完存储程序之后，可以把 mysql 程序的语句终止符重新定义为分号。下面的例子在定义一个存储过程时把 mysql 程序的默认分隔符临时改变为$，然后在恢复了 mysql 程序的默认分隔符之后执行了那个存储过程：

```
mysql> delimiter $
mysql> CREATE PROCEDURE show_times()
    -> BEGIN
    ->   SELECT 'Local time is:', CURRENT_TIMESTAMP;
    ->   SELECT 'UTC time is:', UTC_TIMESTAMP;
    -> END$
mysql> delimiter ;
mysql> CALL show_times();
+----------------+---------------------+
| Local time is: | CURRENT_TIMESTAMP   |
+----------------+---------------------+
| Local time is: | 2008-05-15 18:20:13 |
+----------------+---------------------+
```

```
+--------------+---------------------+
| UTC time is: | UTC_TIMESTAMP       |
+--------------+---------------------+
| UTC time is: | 2008-05-15 23:20:13 |
+--------------+---------------------+
```

分隔符不必非得是$字符，也不必非得是单个的字符：

```
mysql> delimiter EOF
mysql> CREATE PROCEDURE show_times()
    -> BEGIN
    ->   SELECT 'Local time is:', CURRENT_TIMESTAMP;
    ->   SELECT 'UTC time is:', UTC_TIMESTAMP;
    -> END EOF
mysql> delimiter ;
```

这里的原则是：只要在某个存储程序内部的语句里会用到分号，就应该在定义这个存储程序时临时改变 mysql 程序的分隔符。

复合语句并非只能用在复杂的存储程序里。即使你的存储程序只包含一条语句、甚至是没有包含任何语句，也可以使用复合语句：

```
CREATE PROCEDURE do_little ()
BEGIN
  DO SLEEP(1);
END;

CREATE PROCEDURE do_nothing ()
BEGIN
END;
```

为了保持编程风格的统一，建议大家在所有的存储程序定义里使用 BEGIN 和 END。

4.2 存储函数和存储过程

存储函数将向调用者返回一个计算结果，这个结果可以用在表达式里（就像 COS()或 HEX()这样的内建函数那样）。存储过程需要使用 CALL 语句来调用，是一个独立的操作，不能用在表达式里。使用存储过程的情况主要有两种：（1）只需通过运算来实现某种效果或动作而无需返回一个值，（2）运算会返回多个结果集（函数做不到这一点）。这只是些指导性建议，不是硬性规定。比如说，如果你需要返回两个或更多的值，就不能使用函数。但你可以使用一个过程，因为过程支持的参数类型允许它们的值在过程执行期间被设置，而调用者可以在过程返回后去访问那些值。

存储函数要用 CREATE FUNCTION 语句来创建，存储过程要用 CREATE PROCEDURE 语句来创建。下面的例子将创建一个函数，该函数有一个代表着年份的整数参数。（为了与数据表或数据列的名字有所区别，给参数起名字时将使用 p_前缀。）这个函数使用了一个子查询来确定有多少位总统是出生在给定年份里的，并返回这个计数值：

```
mysql> delimiter $
mysql> CREATE FUNCTION count_born_in_year(p_year INT)
    -> RETURNS INT
    -> READS SQL DATA
    -> BEGIN
    ->  RETURN (SELECT COUNT(*) FROM president WHERE YEAR(birth) = p_year);
```

```
    -> END$
mysql> delimiter ;
```

这个函数有一条用来表明其返回值数据类型的 RETURNS 子句和一个用来计算那个值的函数体。函数体至少需要包含一条 RETURN 语句，用来向调用者返回一个值。把计算定义为函数的好处是可以方便地执行它而无须每次都写出所有的逻辑，你可以像使用内建函数那样来调用存储函数：

```
mysql> SELECT count_born_in_year(1908);
+--------------------------+
| count_born_in_year(1908) |
+--------------------------+
|                        1 |
+--------------------------+
mysql> SELECT count_born_in_year(1913);
+--------------------------+
| count_born_in_year(1913) |
+--------------------------+
|                        2 |
+--------------------------+
```

具体到上面的例子，我们在 SELECT 语句里直接调用了 count_born_in_year()函数，但存储函数其实可以用在任意复杂的表达式里。

你无法让一个给定的函数返回多个值。你可以编写任意多个函数，然后在同一条语句里调用它们全体。另一个办法是使用一个存储过程并通过它的 OUT 参数“返回”多个值。存储过程负责计算那些值并把它们赋值给相应的参数，而那些参数可以在过程返回后由调用者访问。这方面的细节请参见4.2.2节。

如果你定义了一个与某个 MySQL 内建函数同名的存储函数，在调用它时就必须用数据库的名字对该函数的名字进行限定以避免歧义。比如说，如果你在 sampdb 数据库里定义了一个名为 PI()的存储函数，就必须使用 sampdb.PI()来调用它以明确地表明打算调用的不是那个同名的内建函数。（为了避免这种麻烦，最好的办法是不要使用内建函数的名字来命名你的存储函数。）

存储过程和存储函数很相似，但它不返回值。因此，它没有 RETURNS 子句或任何 RETURN 语句。下面这个简单的存储过程和 count_born_in_year()函数很相似，它将显示一个结果集而不是把计算结果作为其返回值。那个结果集里的数据行分别对应着在给定年份出生的每一位总统：

```
mysql> delimiter $
mysql> CREATE PROCEDURE show_born_in_year(p_year INT)
    -> BEGIN
    ->   SELECT first_name, last_name, birth, death
    ->     FROM president
    ->     WHERE YEAR(birth) = p_year;
    -> END$
mysql> delimiter ;
```

与存储函数不同，存储过程不能用在表达式里，它们只能通过 CALL 语句来调用。如下所示：

```
mysql> CALL show_born_in_year(1908);
+------------+-----------+------------+------------+
| first_name | last_name | birth      | death      |
+------------+-----------+------------+------------+
| Lyndon B.  | Johnson   | 1908-08-27 | 1973-01-22 |
+------------+-----------+------------+------------+
```

```
mysql> CALL show_born_in_year(1913);
+------------+-----------+------------+------------+
| first_name | last_name | birth      | death      |
+------------+-----------+------------+------------+
| Richard M. | Nixon     | 1913-01-09 | 1994-04-22 |
| Gerald R.  | Ford      | 1913-07-14 | 2006-12-26 |
+------------+-----------+------------+------------+
```

对于本例，过程体执行了一条 SELECT 语句。这个例子表明，结果集没有被返回为过程值，而是被发送给了客户。一个过程可以生成多个结果集，它们将依次被发送到客户。

前面的例子都是选取信息，但存储例程还可以用来修改数据表，如下例所示。update_expiration()是一个用来更新数据的存储例程。它将根据一个历史研究会会员的 ID 号用一个给定的失效日期去更新相应的会员资格数据行：

```
CREATE PROCEDURE update_expiration (p_id INT UNSIGNED, p_date DATE)
BEGIN
  UPDATE member SET expiration = p_date WHERE member_id = p_id;
END;
```

下面的 update_expiration()调用将把会员资格失效日期设置为明年的今天或者将会员设置为“永久会员”（NULL 表示“永不失效”）：

```
mysql> CALL update_expiration(61, CURDATE() + INTERVAL 1 YEAR);
mysql> CALL update_expiration(87, NULL);
```

存储函数必须遵守这样一条限制：不允许对调用本函数的语句正在读或写的数据表进行修改。存储过程通常没有这个限制，但如果它们是从存储函数里被调用，就需要遵守这条限制。比如说，如果一条从 member 数据表选取数据的语句使用了一个存储函数，从这个存储函数里调用 update_expiration()过程将是不允许的。

4.2.1 存储函数和存储过程的权限

存储函数和存储过程属于数据库。要想创建存储函数或存储过程，必须拥有那个数据库的 CREATE ROUTINE 权限。在默认的情况下，当你创建一个存储例程时，服务器将自动地把 EXECUTE 和 ALTER ROUTINE 权限授予你（如果你还没有获得这些权限），这样你才可以执行那个例程或删除它。当你删除那个例程时，服务器将自动撤销那些权限。如果你不想使用这种自动化的权限授予/撤销机制，把 automatic_sp_privileges 系统变量设置为 0 即可。

如果服务器启用了二进制日志功能，存储函数还需要遵守一些额外的限制条件（不允许创建不确定或是会修改数据的存储函数）以保证二进制日志能够安全地完成备份和复制操作。（如果某个函数会为给定的输入值生成不同的结果，通过重新执行二进制日志而恢复数据的办法将不能保证把数据恢复到原来的样子，而且该函数还可能导致在主服务器和从服务器上的复制结果不一致。）这些限制条件如下所示。

- 如果 log_bin_trust_function_creators 系统变量没有被激活，你就必须具备 SUPER 权限才能创建存储函数。在此前提下，你创建的每一个函数都必须是确定的，并且不得修改数据。为了表明这一点，需要使用 DETERMINISTIC、NO SQL 或 READS SQL DATA 之一来定义存储函数。例如：

```
CREATE FUNCTION half (p_value DOUBLE)
RETURNS DOUBLE
DETERMINISTIC
BEGIN
  RETURN p_value / 2;
END;
```

- 如果 log_bin_trust_function_creators 系统变量已被激活，则没有任何限制。只有当你可以相信 MySQL 服务器上的所有用户都不会去定义不安全的存储函数时，这种设置才是最适当的。

与 log_bin_trust_function_creators 系统变量有关的限制条件同样适用于触发器的创建工作。在 MySQL 5.1.6 版本之前，你不太可能注意这一点，因为你必须拥有 SUPER 权限才能创建触发器，而 SUPER 权限覆盖了与 log_bin_trust_function_creators 系统变量有关的限制条件。

4.2.2　存储过程的参数类型

存储过程的参数分为 3 种类型。对于 IN 参数，调用者把一个值传递给过程，过程可以对这个值进行修改，但任何修改在过程返回后对调用者是不可见的。OUT 参数刚好相反，过程把一个值赋值给 OUT 参数，这个值在过程返回后可以由调用者访问。INOUT 参数允许调用者向过程传递一个值，然后再取回一个值。

要想明确地为参数指定类型，在参数表里把 IN、OUT 或 INOUT 写在参数名字前面即可。如果没有为参数指定类型，其默认类型将是 IN。

在使用 OUT 或 INOUT 参数时，在调用过程时需要给出一个变量名。过程可以设置参数的值，相应的变量将在过程返回时获得那个值。如果想让某个存储过程返回多个结果值，OUT 和 INOUT 参数类型将非常有用（存储函数只能返回一个值，不能胜任）。

下面的过程演示了 OUT 参数的用法。它将分别统计出 student 数据表里的男生和女生人数并通过它的参数返回这两个计数值，让调用者可以访问它们：

```
CREATE PROCEDURE count_students_by_sex (OUT p_male INT, OUT p_female INT)
BEGIN
  SELECT COUNT(*) FROM student WHERE sex = 'M' INTO p_male;
  SELECT COUNT(*) FROM student WHERE sex = 'F' INTO p_female;
END;
```

在调用这个过程时，请把各个参数替换为相应的用户定义变量。这个过程将把计数值放到这些参数里，在它返回之后，那些变量将包含计数值：

```
mysql> CALL count_students_by_sex(@mcount, @fcount);
mysql> SELECT 'Number of male students:  ', @mcount;
+----------------------------+---------+
| Number of male students:   | @mcount |
+----------------------------+---------+
| Number of male students:   |      16 |
+----------------------------+---------+
mysql> SELECT 'Number of female students:', @fcount;
+----------------------------+---------+
| Number of female students: | @fcount |
+----------------------------+---------+
| Number of female students: |      15 |
+----------------------------+---------+
```

更复杂的处理可能需要更多的参数。比如说，你可以编写一个这样的过程：用 IN 参数来获得 score 数据表里的某次考试或测验的 ID 编号，在过程里对相关考试成绩进行各种统计分析（平均值、标准偏差、范围等），然后把所有那些值通过 OUT 参数返回给调用者。

IN、OUT 和 INOUT 关键字不适用于存储函数、触发器或事件。对于存储函数，所有的参数都像 IN 参数。触发器和事件则根本没有任何参数。

4.3 触发器

触发器是与特定数据表相关联的存储过程，当相应的数据表被 INSERT、DELETE 或 UPDATE 语句修改时，触发器将自动执行。触发器可以被设置成在这几种语句处理每个数据行之前或之后触发。触发器的定义包括一条将在触发器被触发时执行的语句。

4

下面描述了触发器提供的一些好处。

- 触发器可以检查或修改将被插入或用来更新数据行的新数据值。这意味着我们可以利用触发器强制实现数据的完整性，比如检查某个百分比数值是不是落在了 0 到 100 的区间内。触发器还可以用来对输入数据进行必要的过滤。
- 触发器可以把表达式的结果赋值给数据列作为其默认值。这使我们可以绕开数据列定义里的默认值必须是常数的限制。
- 触发器可以在删除或修改数据行之前先检查它的当前内容。这种能力可以用来实现许多功能，例如把对现有数据行的修改记载到一个日志里。

触发器要用 CREATE TRIGGER 语句来创建。在触发器的定义里需要表明它将由哪种语句（INSERT、UPDATE 或 DELETE）触发，是在数据行被修改之前还是之后被触发。触发器创建语句的基本语法如下所示：

```
CREATE TRIGGER trigger_name        # the trigger name
  {BEFORE | AFTER}                 # when the trigger activates
  {INSERT | UPDATE | DELETE}       # what statement activates it
  ON tbl_name                      # the associated table
  FOR EACH ROW trigger_stmt;       # what the trigger does
```

tb_name 是与触发器相关联的数据表的名字；trigger_name 是触发器本身的名字。在给触发器起名时，我喜欢把触发器的用途和数据表的名字包括在其中，比如说，bi_tbl_name 表示这是一个与 tbl_name 数据表相关联的 BEFORE INSERT 触发器，ai_tbl_name 表示这是一个与 tbl_name 数据表相关联的 AFTER INSERT 触发器。

trigger_stmt 是触发器的语句体部分，也就是在触发器被触发时将要执行的语句。在触发器的语句体里，可以使用 NEW.col_name 语法来引用将由 INSERT 或 UPDATE 语句插入或修改的新数据行里的数据列。类似地，OLD.col_name 语法可以用来引用将由 DELETE 或 UPDATE 语句删除或修改的老数据行里的数据列。比如说，如果你想在新数据行被插入数据表之前改变它的某个数据列的值，只需创建一个 BEFORE 触发器并在其语句体里写出相应的“SET NEW.col_name =value”语句即可。

在下面的例子里，我们为数据表 t 上的 INSERT 语句创建了一个名为 bi_t 的触发器。数据表 t 有一个整数类型的 percent 数据列用来保存百分比数值（0 到 100）和一个 DATETIME 数据列。我们在定义这个触发器时使用了 BEFORE 关键字，所以它将在数据值被插入数据表之前对它们进行检查：

```
mysql> CREATE TABLE t (percent INT, dt DATETIME);
```

```
mysql> delimiter $
mysql> CREATE TRIGGER bi_t BEFORE INSERT ON t
    ->   FOR EACH ROW BEGIN
    ->     SET NEW.dt = CURRENT_TIMESTAMP;
    ->     IF NEW.percent < 0 THEN
    ->       SET NEW.percent = 0;
    ->     ELSEIF NEW.percent > 100 THEN
    ->       SET NEW.percent = 100;
    ->     END IF;
    ->   END$
mysql> delimiter ;
```

这个触发器将完成如下两个动作。

- 如果将被插入的百分比值超出了 0 到 100 的范围，这个触发器将把该值转换为最近的边界值（0 或 100）。
- 这个触发器将自动地为那个 DATETIME 数据列提供一个 CURRENT_TIMESTAMP 值。从效果上看，这绕开了“数据列的默认值必须是一个常数”的限制，让 DATETIME 数据列具备了类似于 TIMESTAMP 数据列的自动初始化能力。

我们来看看这个触发器的工作情况。先在数据表里插入一些数据行，然后再检索它的内容：

```
mysql> INSERT INTO t (percent) VALUES(-2); DO SLEEP(2);
mysql> INSERT INTO t (percent) VALUES(30); DO SLEEP(2);
mysql> INSERT INTO t (percent) VALUES(120);
mysql> SELECT * FROM t;
+---------+---------------------+
| percent | dt                  |
+---------+---------------------+
|       0 | 2008-05-15 18:38:22 |
|      30 | 2008-05-15 18:38:24 |
|     100 | 2008-05-15 18:38:26 |
+---------+---------------------+
```

创建和删除触发器所需要的权限取决于具体的 MySQL 版本。在 MySQL 5.1.6 版之前，你必须拥有 SUPER 权限。自 MySQL 5.1.6 版起，访问控制得到了更正确的处理：因为触发器与一个数据表相关联，所以你必须拥有那个数据表的 TRIGGER 权限才能为它创建和删除触发器。

4.4 事件

MySQL 5.1.6 及更高版本有一个事件调度器，它使我们可以把数据库操作安排在预定时间执行。事件是与一个时间表相关联的存储程序，时间表用来定义事件发生的时间、次数以及何时消失。事件非常适合用来执行各种无人值守的系统管理任务，如定期更新汇总报告、清理过期失效的数据、对日志数据表进行轮转等。本节演示如何对过期失效的数据行进行处理。利用事件对日志数据表进行轮转的例子请参见 12.5.7 节中的第 4 小节。

在默认的情况下，事件调度器不会运行。如果你想使用事件，必须先启用事件调度器。把以下语句添加到一个选项文件中（服务器在启动时将读取）：

```
[mysqld]
event_scheduler=ON
```

如果你想在系统运行时查看事件调度器的状态，可以使用这条语句：

```
SHOW VARIABLES LIKE 'event_scheduler';
```

如果你想在系统运行时停止或启动事件调度器，可以通过改变 event_scheduler 系统变量的值来达到目的（它是一个 GLOBAL 变量，你必须拥有 SUPER 权限才能修改它）：

```
SET GLOBAL event_scheduler = OFF;     # or 0
SET GLOBAL event_scheduler = ON;      # or 1
```

如果你停止了事件调度器，就没有事件可以运行了。你也可以让事件调度器保持运行，但禁用各事件稍后将讨论。

说明 如果你在启动服务器时把 event_scheduler 变量设置为 DISABLED，在系统运行时，你将只能查看它的状态，不能改变它的状态。在此基础上，你仍可以创建事件，但它们将不能执行。

事件调度器将把它的执行情况写到服务器的“错误”日志里，你可以从这个日志查到关于事件调度器正在干什么的信息。它会把它运行的每一个事件以及在事件执行过程中发生的错误记载到日志里。如果你认为事件调度器应该正在运行、可它实际上并没有运行，请到“错误”日志里找找原因。

下面的例子演示了如何创建一个简单的事件来删除数据表里“老”数据行。假设你有一个名为 web_session 的数据表，其内容是访问你网站的用户的会话状态信息。这个数据表有一个名为 last_visit 的 DATETIME 数据列，记录着每位用户最近一次的访问时间。如果不想让这个数据表里的老数据行越积越多，就创建一个事件来定期清理它们。若要让这个事件每隔 4 小时执行一次，就把超过一天的数据行清除掉。下面是相应的事件定义：

```
CREATE EVENT expire_web_session
  ON SCHEDULE EVERY 4 HOUR
  DO
    DELETE FROM web_session
    WHERE last_visit < CURRENT_TIMESTAMP - INTERVAL 1 DAY;
```

EVERY n interval 子句用来给出事件定期执行的时间间隔。interval 值近似于 DATE_ADD()函数里的参数值，它可以是 HOUR、DAY 或 MONTH。在 EVERY 子句的后面，你还可以用 STARTS datetime 和 ENDS datetime 选项给出事件第一次和最后一次执行的时间。在默认情况下，EVERY 事件将在它被创建后立刻开始它的第一次执行，并将一直定期执行下去，没有“最后一次”的说法。

DO 子句负责定义事件的语句体部分，也就是将在事件被触发时执行的 SQL 语句。和其他类型的存储程序一样，这可以是一条简单的语句，也可以是一条以 BEGIN 开始、以 END 结束的复合语句。

如果想创建一个只执行一次的事件，就应该使用 AT 调度类型而不是 EVERY。如下所示的定义将创建一个只执行一次的事件，在一个小时后执行：

```
CREATE EVENT one_shot
  ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 HOUR
  DO ... ;
```

如果你想禁用某个事件，让它不再定期执行，或者重新激活某个已被禁用的事件，请使用 ALTER EVENT 语句：

```
ALTER EVENT event_name DISABLE;
ALTER EVENT event_name ENABLE;
```

每个事件都隶属于某个数据库，所以你必须拥有那个数据库的 EVENT 权限才能为它创建或删除事件。

4.5　存储程序和视图的安全性

创建一个存储程序，也就创建了一个将在未来的某个时刻执行的对象。定义视图时也是如此，包含在视图定义里的 SELECT 语句同样将在未来的某个时刻执行。既然是“在未来的某个时刻执行”，就意味着实际执行这些语句的用户有可能不是当初创建它们的那位。这就引出了一个重要的问题：服务器在执行时应该使用什么样的安全上下文来检查访问权限？换个问法，应该使用哪个账户的权限？

在默认情况下，服务器将使用当初定义对象的那位用户的账户。假设我定义了一个存储过程 p()来访问一个属于我的数据表。如果我把 p()的 EXECUTE 权限授予了你，你就可以使用 CALL p()来调用该过程去访问我的数据表，因为它是以我的权限运行的。这样的安全上下文安排既有好处，又有坏处。

- 好处是：精心编写的存储程序可以把数据表开放给那些无权直接访问它们的用户，而开放程度由存储程序的创建者控制。
- 坏处是：如果某个用户创建了一个存储程序来访问敏感的数据，可以调用这个对象的其他人将获得与该对象的定义者同样的数据访问权限。

在定义存储程序或视图时，你可以通过在 CREATE 语句里使用一个 DEFINER = account 子句明确指定该存储程序或视图的定义者。这将使你给定的账户被当做该存储程序或视图的定义者，MySQL 在执行该存储程序或视图时将使用该账户去核查访问权限。比如说：

```
CREATE DEFINER = 'sampadm'@'localhost' PROCEDURE count_students()
  SELECT COUNT(*) FROM student;
```

在 DEFINER 子句里，DEFINER 值可以是'user_name'@'host_name'格式的账户名，CREATE USER 等账户管理语句里的账户名也是这种格式（参见 12.4.1 节中的第 1 小节）。如果使用这种格式，user_name 和 host_name 都必不可少。另外，这个值还可以是 CURRENT_USER 或 CURRENT_USER()，这表明定义者就是执行本语句的用户的账户（如果没有给出 DEFINER 子句，则默认使用同一个账户）。

如果你有 SUPER 权限，就可以使用任何一个语法正确的账户名作为 DEFINER 值。如果这个账户名当时尚不存在，MySQL 将生成一条警告消息。如果你没有 SUPER 权限，就只能把 DEFINER 设置为你自己的账户，只要使用完整的账户名或 CURRENT_USER 均可。

对于视图、存储函数和存储过程，你还可以给出 SQL SECURITY 选项，以另一种方式控制 MySQL 在执行期间对视图/存储函数/存储过程的访问权限检查。SQL SECURITY 选项的可取值是 DEFINER（意思是“以定义者的权限执行”）或 INVOKER（意思是“以调用者的权限执行”）。

SQL SECURITY INVOKER 最适合这样的场合：你不想让存储程序或视图在执行时的权限多于其调用者。例如，下面的视图将访问 mysql 数据库里的一个数据表，但只能以调用者的权限运行。这样一来，如果调用者本人无权访问 mysql.user 数据表，即使他执行了这个视图也不会看到他不应该看到的东西。

```
CREATE SQL SECURITY INVOKER VIEW v
  AS SELECT CONCAT(User,'@',Host) AS Account, Password FROM mysql.user;
```

触发器和事件是由服务器自动调用的，所以“调用者”的概念不适用于它们。它们不支持 SQL SECURITY 选项，它们总是以定义者的权限执行。

如果把一个存储程序或视图定义为以 DEFINER 的权限运行，但在其定义里 DEFINER 账户并不存在，MySQL 在执行该存储程序或视图时将抛出一个错误。

第5章

查询优化

关系数据库的理论就是关于数据表、集合以及对它们进行操作的理论。数据库是数据表的集合，数据表是数据行和数据列的集合。当你发出一个 SELECT 语句来从数据表中检索数据行时，你得到的是另外一个数据行和数据列的集合，也就是另一个数据表。这都是一些抽象的概念，不涉及数据库系统用来操作数据表里的数据的底层表示。另外一个抽象的概念是，对数据表的操作总是同时发生的。查询在概念上是集合操作，因而在集合理论中没有关于时间的概念。

当然，真实的情况是大不相同的。数据库管理系统的确能实现抽象的概念，但实现这些概念靠的是真实有形的硬件设备。因此，查询要占用时间，有时甚至长得令人厌烦。而急躁的我们可不喜欢等待，于是我们把对集合瞬时准确操作的抽象概念放在一边，继而寻找能够加快查询的方法。不过，还真有一些方法可以实现这个目标：

- 为数据表创建索引以使数据库服务器能更快地查阅数据行；
- 看看如何写出查询以最大程度地利用那些索引，使用 EXPLAIN 语句检查 MySQL 服务器是否真的如期行事；
- 编写查询来影响服务器的调度机制，从而使来自多个客户程序的查询能够更好地协作；
- 修改服务器的操作参数以提高它的工作效率；
- 分析底层硬件在做些什么，分析如何解决物理限制，从而提高性能。

上面这些方面本章要着重讨论的内容，目的是帮助你优化数据库系统的性能，使它尽可能快地处理你的查询。MySQL 数据库已经很快了，但即使运行查询最快的数据库，你也可以通过优化使它运行得更快。

5.1 使用索引

用来加快查询的技术有很多，其中最重要的是索引。通常，能够造成查询速度最大差异的是索引的正确使用。很多时候，当查询速度很慢时，添加上索引后就能迅速解决问题。但情况也不总是这样，因为优化并不总是一件简单的事情。然而，在许多情况下，假如你不使用索引，那么试图通过其他途径来提高性能则纯粹是浪费时间。你应该首先使用索引来最大程度地改进性能，然后再看是否还有其他技术可以采用。

本节讨论索引是什么以及索引是怎样改进查询性能的，还会讨论索引可能降低性能的情况，还会对于怎样为数据表合理地选择索引提供指导。在下一节中，我们将讨论 MySQL 的查询优化程序，这种程序用于找到执行查询的最有效途径。除了掌握如何创建索引以外，再具备一些优化程序方面的知

识是有益处的，这样一来，你就能更充分地利用你所创建的索引。有些编写查询的方法实际上阻止了你有效地利用它们，通常你会希望避免发生这种情况。

5.1.1　索引的优点

让我们一起来讨论索引是如何工作的，首先介绍没有索引的数据表。一个没有索引的数据表就是一个无序的数据行集合。图 5-1 是一个 ad 数据表，我们在第 1 章就已经看到过了。现在在这个数据表上没有索引，如果我们想找到一个公司的数据行，就需要检查数据表的每一个数据行，看它是否与期望值匹配。这是一个数据表的完整扫描，如果数据表很大，但是仅有少数几个记录与搜索条件相匹配，那么工作过程就很慢，效率很低。

ad数据表

company_num	ad_num	hit_fee
14	48	0.01
23	49	0.02
17	52	0.01
13	55	0.03
23	62	0.02
23	63	0.01
23	64	0.02
13	77	0.03
23	99	0.03
14	101	0.01
13	102	0.01
17	119	0.02

图 5-1　没有索引的 ad 数据表

图 5-2 是同一个数据表，但是在 ad 数据表的 company_num 数据列上添加了索引。索引中包含了数据表里每一个数据行的项，而且项是根据 company_num 值来分类的。现在，就不用再一行一行地搜索整个数据表来寻找匹配项了，我们可以使用索引。假定我们正在搜寻的是公司编号为 13 的所有数据行。我们开始扫描索引并找到了 3 个匹配的数据行。然后我们到达公司编号为 14 的数据行，这个值高于我们要搜寻的值。由于索引值是经过分类的，所以，当我们读到包含 14 的记录时，我们就知道，不会再有与 13 相匹配的内容了，因此不再扫描。由此可见，索引可以提高搜索效率的一个原因就是我们可以得知匹配数据行在什么位置结束，从而跳过其余部分。另外一个原因则是定位算法的使用，它们可以不用从索引开始位置经过线性扫描就能直接就找到第一个匹配项（例如，二进制搜索就比扫描快许多）。使用这种方式，我们可以迅速地到达第一个匹配值，从而节省了大量搜索时间。各种数据库使用着各种各样的技术来迅速地找到索引值，但在这里，我们不着重讨论这些技术方法。重要的是我们要知道这些技术方法可以加快索引速度，并且要知道索引的确是个很好的方法。

你可能会问，为什么不直接将数据行分类，从而省掉索引呢？这样做不也能够加快搜索速度吗？假如数据表只有一个索引的话，回答是肯定的。但是你有可能想添加上第二个索引，但又不能同时将这个数据行按照两种方式分类。（例如，需要用到一个关于顾客姓名的索引，还有一个是顾客 ID 号的索引或电话号码的索引。）将索引从数据行中整体分出就可以解决这个问题，可以创建多个索引。此外，索引中的数据行通常都要比数据文件的数据行短一些。当你插入或删除数值时，来回移动较短的索引数值来保持分类顺序要比来回移动较长的数据文件的数据行方便一些。

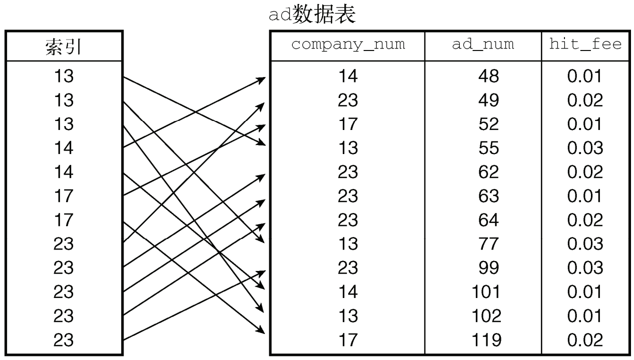


图 5-2 有索引的 ad 数据表

对于不同的存储引擎，索引实现的细节有所不同。例如，对于 MyISAM 数据表来说，数据表的数据行是在数据文件里，而索引值是在索引文件里。一个数据表可以有多个索引，但如果这样的话，所有的索引都储存在同一个索引文件里。索引文件里的每一个索引都是由分类的关键记录数组组成的，这些数组用于快速访问数据文件。

相比之下，InnoDB 存储引擎没有按照上面的方法将数据行和索引值分开，尽管它们也都保持着索引数值分类放置的特点。默认情况下，InnoDB 存储引擎使用的是一个表空间，在这个表空间里，它管理着所有 InnoDB 类型数据表的数据和索引的存储。我们可以通过配置让 InnoDB 为每个数据表分别创建一个它自己的表空间，但即便如此，一个给定的数据表的数据和索引也是保存在同一个表空间文件里的。

上面的讨论描述了在单个数据表查询的情况（使用索引可以不再需要完成全数据表扫描，从而极大地提高了搜索的速度）下索引所具有的优点。在你运行联结多个数据表的查询时，索引实际上可以发挥更大的作用。在单个数据表的查询里，你需要就每一个数据列检查的数值数目就是数据表里数据行的个数。在多个数据表的查询里，这个数目可能是一个天文数字，因为它是这些数据表中所有数据行的个数。

假定你有 3 个没有索引的数据表 `t1`、`t2` 和 `t3`，每个数据表都包含一个数据列 `c1`、`c2` 和 `c3`，而且每一个数据列都有从数字 1 到数字 1000 的 1000 个数据行。要找出这些数据表中具有相同数值的所有数据行的组合，其查询语句应该是下面这样：

```
SELECT t1.i1, t2.i2, t3.i3
FROM t1 INNER JOIN t2 INNER JOIN t3
WHERE t1.i1 = t2.i2 AND t2.i2 = t3.i3;
```

这个查询的结果应该有 1000 个数据行，每行都包含 3 个相等的数值。假如我们不使用索引来查询，而且也不全部扫描，根本就不知道哪些数据行包含哪些数值。因此，我们必须组合所有的这些数据行来找到哪些数据行与 `WHERE` 子句相匹配。可能的组合有 1000×1000×1000（10 亿）种，比匹配数目多 100 万倍。这可是一种极大的浪费。当数据表数目增加时，如果不使用索引，处理这些数据表的联结情况所需要的时间会增加更多，导致性能更差。如果我们为数据表编制索引，我们就能很大程度地提高查询速度，因为利用索引可以像下面这样处理查询。

(1) 从数据表 `t1` 中选择第一个数据行，看这个数据行包含什么样的值。

(2) 对数据表 `t2` 使用索引，直接找到与数据表 `t1` 的值相匹配的数据行。类似地，对数据表 `t3` 使

用索引，直接找到与数据表 t1 的值相匹配的数据行。

(3) 对数据表 t1 的下一个数据行重复上面的过程，直到检查完数据表 t1 的所有数据行。

在这个例子中，我们仍然对数据表 t1 完成了全扫描的过程，但是我们能够对 t2 和 t3 进行带索引的搜寻，直接将那些数据行挑选出来。这种查询的运行速度比不使用索引的方式快 100 万倍，这可是一点儿都不夸张。当然，这个例子只不过是用来说明索引的用处的。然而，它所揭示的问题却是真实的，为一个没有索引的数据表加上索引通常能够使性能得到巨大改进。

MySQL 使用索引的方式有以下几种。

- 如上所述，索引的用途，一是在查询操作中把与 WHERE 子句所给出的条件相匹配的数据行尽快找出来；二是在关联操作中把与其他数据表里的数据行相匹配的数据行尽快找出来。
- 对于使用 MIN( )或 MAX( )函数的查询，如果数据列带索引，那么它的最小值和最大值能够被迅速找到而不用通过逐行检查的方法来查找。
- MySQL 经常使用索引来迅速地完成 ORDER BY 子句和 GROUP BY 子句的分类和分组操作。
- 有时，MySQL 可以通过使用索引来避免为一个查询整体读取数据行。假如你是从 MyISAM 数据表的一个有索引的数字数据列里选取值，而且你并不打算选取数据表的其他数据列。在这种情况下，MySQL 从这个索引文件读取索引值时，你实际上就已经得到了这个值，而你本来是应该通过读取数据文件才能得到的。不必读取两次值，因此，甚至都没有必要参考数据文件。

5.1.2　索引的缺点

一般来说，如果 MySQL 知道如何通过使用索引来更快地处理查询，它就会使用索引。这也就意味着，在绝大多数情况下，如果你不为数据表编制索引，你简直就是在伤害自己。你可以看到我已经就索引的优点描绘出了一幅玫瑰花般美丽的图片。那它们有没有缺点呢？有的，它们也有一些时间和空间上的缺点。在实际运用中，这些缺点被优点掩盖住了，但是你应该知道这些缺点是什么。

首先，索引加快了检索速度，但是却降低了在带索引的数据列里插入、删除以及修改数值的速度。也就是说，索引降低了许多涉及写入的操作的速度。之所以会出现这种情况，是由于写入一条数据行不仅要求写入到数据行，它还要求所有索引都要做出改变。一个数据表有越多的索引，需要做出的改变就越多，平均性能下降就越多。绝大部分数据表是读操作多、写操作少，但对那些写操作次数比较多的数据表来说，索引更新方面的开销可能会非常大。5.4 节将讨论如何减少这方面的开销。

其次，索引要占据磁盘空间，多个索引会占据更大的空间。与没有索引相比较，这会使你较快地到达数据表的尺寸极限。

- 对 MyISAM 类型的数据表来说，大量地索引一个数据表有可能使索引文件比数据文件更快地到达它的最大尺寸。
- 存储在 InnoDB 共享表空间里的全部 InnoDB 数据表分享着同一个存储空间，添加索引会使表空间里用于存储的空间更快地减少。和用于 MyISAM 数据表的文件不同，InnoDB 数据表的共享表空间不受操作系统文件尺寸的限制，因为它可以包含多个文件。如果你有一个附加磁盘空间，你就可以对它添加新的部件来扩充表空间。

  使用单独表空间的 InnoDB 数据表把数据和索引集中保存在同一个文件里，增加索引将导致数据表的尺寸更快地逼近最大文件长度。

上述两个因素的现实指导意义是：如果不需要某个特定的索引来加快查询速度，就不要创建它。

5.1.3 挑选索引

关于创建索引的语法已经在 2.6.4 节中的第 2 小节里进行了论述。我想你们已经读过了那一节的内容。但是仅仅知道语法并不能帮助你掌握对数据表编制索引，要掌握这些内容，还需要讨论一些使用数据表的方式的知识。这一节给出了关于如何确定要索引的数据列以及如何正确地建立索引的一些准则。

尽量为用来搜索、分类或分组的数据列编制索引，不要为作为输出显示的数据列编制索引。换句话说，最适合有索引的数据列是那些在 WHERE 子句中出现的数据列、在联结子句中给出的数据列，或者是在 ORDER By 或 GROUP BY 子句中出现的数据列。根据 SELECT 关键字仅出现在输出数据列清单里的数据列最好不要有索引：

```
SELECT
    col_a                          ← not a candidate
FROM
    tbl1 LEFT JOIN tbl2
    ON tbl1.col_b = tbl2.col_c     ← candidates
WHERE
    col_d = expr;                  ← a candidate
```

当然，你显示的数据列和你在 WHERE 子句中使用的数据列可能会一样。问题在于输出数据列表单中出现的数据列本身并不能说明它应该有索引。

出现在联结子句中的数据列，或者 WHERE 子句中 col1 = col2 格式的表达式里的数据列特别适合用于索引。上面查询中给出的 col_b 和 col_c 就是这样的例子。如果 MySQL 能够使用相联结的数据列来优化查询，它就不会使用全数据表扫描，这样能删除掉大量潜在的数据表–数据行组合。

综合考虑各数据列的维度势。数据列的“维度”（cardinality）等于它所容纳的非重复值的个数。比如说，如果某个数据列里的值分别是 1、3、7、4、7、3，它的维度就是 4。数据列的维度越高（维度的最大值等于数据表里的数据行的个数），它包含的独一无二的值就越多，重复的值就越少，索引的使用效果也就越好。举例来看，如果一个数据列包含许多不同的年龄值，索引将迅速地将数据行区分开来。但是对于记录性别的数据列，其中只有两个值“M”和“F”，索引恐怕就帮不了你了。如果这两个值出现情况大致一样多，那么不管你搜索的是哪一个数值，你都会得到半数左右的数据行。在这种情况下，索引或许就根本不能够使用，因为当查询优化程序确定出某一个数值在数据表的数据行中出现频率超过 30%时，查询优化程序通常会跳过索引，而进行全数据表扫描。现如今的优化器更复杂，能够把其他因素也考虑进来，所以这个百分比已经不再是 MySQL 决定进行一次扫描而不是使用一个索引的唯一依据了。

对短小的值进行索引。应尽量选用比较“小”的数据类型。比如说，如果一个 MEDIUMINT 数据列已足以容纳你需要存储的数据，就不要选用 BIGINT；如果你的数据没有一个比 25 个字符更长，就不要选用 CHAR(100)。比较短小的值可以在以下几个方面提高索引的处理性能。

- 短小的值可以让比较操作更快地完成，加快索引查找速度。
- 短小的值可以让索引的“体积”更小，减少磁盘 I/O 活动。
- 短小的键值意味着键缓存里的索引块可以容纳更多的键值，让 MySQL 可以在内存里同时容纳更多的键，而这将加大在不需要从磁盘读取更多索引块的前提下在内存里找到键值的概率。

对 InnoDB 存储引擎而言，因为它使用的是聚集索引，所以让主键尽量短小将更有好处。所谓“聚

集索引”（clustered index）是指把数据行和主键值集中保存在一起的情况。其他的索引都是些二级索引——它们保存着主键值和二级索引值。先在二级索引里找到一个主键值，再通过它找到相应的数据行。这意味着主键值在每一个二级索引里都会重复出现，如果主键值比较长的话，就会导致每一个二级索引都将需要占用更多的存储空间。

为字符串值的前缀编索引。假如你要为字符串数据列编索引，应当尽可能给出前缀长度。例如，假定你有一个 CHAR(200)的数据列，大多数的值的前 10 个或 20 个字符都是唯一的，那么你就用不着为整个数据列编索引。仅为前面的 20 或 30 个字符编索引可以节省索引中的大量空间，而且会使查询进行得更快。为较小的值编索引可以较少磁盘输入/输出，加快比较速度。当然，你会想继续使用一些惯常的做法。但只为第一个字符编索引恐怕不行，因为这样一来索引中将不会有太多唯一的值。

充分利用最左边的前缀。当你创建了一个 *n* 个数据列的复合索引时，实际上就创建了 MySQL 能够使用的 *n* 个索引。一个复合索引在工作时就相当于 *n* 个索引，因为索引中最左边的数据列集合能够用于匹配数据行。这样的一个集合就称为“最左边的前缀”。（这和为数据列的前缀编索引是不同的，它是将数据列的前 *n* 个字符或字节作为索引值。）

假定你有一个数据表，其数据列有复合索引，数据列名称是 state（州）、city（城市）以及 zip（邮政编码）。索引中的数据行是以州/城市/邮政编码的顺序存储的，因此它们自动地以州/城市的顺序，同时也以州的顺序分类。这就意味着 MySQL 能够充分利用索引，即使你在查询中仅给出了州的值或者仅给出了州和城市的值。因此，索引能够用来搜索下面的数据列的组合：

```
state, city, zip
state, city
state
```

MySQL 不能使用没有包含最左边前缀的搜索的索引。例如，如果你按照城市或邮政编码来搜索，索引就不能使用。如果你搜索的是一个给定的州以及特定的邮政编码（索引的第一个数据列和第三个数据列），尽管这时 MySQL 能够使用索引来找到那些与这个州相匹配的数据行从而缩小搜索范围，但是这个索引不能用于值的组合。

适可而止，不要建立过多的索引。不要以为索引越多越好，不要为你看到的所有东西都编索引，这是一种错误的做法。如上所述，每一个多出的索引都要占据额外的磁盘空间，而且都会影响写入操作的性能。当你修改了数据表的内容后，索引必须要更新以及重新编排，你使用的索引越多，这个过程占据的时间就越长。如果你有一个很少使用或从没有用过的索引，你就无谓地降低了数据表修改的速度。此外，MySQL 在为检索生成一个执行方案时都要仔细对索引进行推敲。创建多余的索引对查询优化程序就加上了更多的工作。而且，当你有太多的索引时，MySQL 还有可能（当然这只是一种可能）无法选出最好的索引来使用。仅保留你需要的索引可以帮助查询优化程序避免造成这样的错误。

如果你想把一个索引添加到一个已经有索引的数据表，要考虑你要添加的索引是否是一个已经存在的多数据列索引的最左边的前缀。如果是这样的话，就不要再费心思添加它了，事实上，你已经有了这个索引。例如，假定你已经有了一个对州、城市和邮政编码的索引，那就没有任何意义再添加一个对州的索引了。有一个例外，对于 FULLTEXT 索引，你要搜索的每个不同的数据列集都必须有自己的索引。

让索引的类型与你打算进行的比较操作的类型保持匹配。在创建索引的时候，绝大多数存储引擎会选择它们将使用的索引实现。比如说，InnoDB 总是使用“B 树”索引。MyISAM 也使用“B 树”索引，但在遇到空间数据类型时会改用“R 树”索引。MEMORY 存储引擎默认使用散列索引，但也支

持“B 树”索引并允许在这两者当中做出选择。在挑选索引类型的时候，一定要考虑你打算在被索引的数据列上进行什么类型的比较操作。

- 对于散列索引，会有一个散列函数来依次处理每一个数据列值。结果散列值将被存入该索引并用来进行查询。（散列函数采用的算法会尽量为不同的输入值生成不同的散列值。使用散列值的好处是它们之间的比较比其原始值更有效率。）散列索引在使用“=”或“<=>”操作符进行的精确匹配比较操作里速度极快。但它们在用来查找一个范围的比较操作里表现不佳，例如下面这些表达式：

```
id < 30
weight BETWEEN 100 AND 150
```

- “B 树”索引在使用<、<=、=、>=、>、<>、!= 和 BETWEEN 操作符进行的精确比较操作或范围比较操作里都很有效率。如果匹配模式是以一个纯字符串而不是一个通配符开头的话，“B 树”索引还可以用于使用 LIKE 操作符进行的模式匹配操作。

如果使用的 MEMORY 数据表只用于精确查找，散列索引会是一个很好的选择。这正是 MEMORY 数据表的默认索引类型，所以不需要做什么特别的事情。如果想用某个 MEMORY 数据表来进行范围比较，就应该为它创建一个“B 树”索引。创建这种索引的具体做法是：在索引定义里加上 USING BTREE 字样。比如说：

```
CREATE TABLE lookup
(
    id      INT NOT NULL,
    name    CHAR(20),
    PRIMARY KEY USING BTREE (id)
) ENGINE = MEMORY;
```

只要你打算使用的搜索类型不挑剔，完全可以让同一个 MEMORY 数据表同时拥有一个散列索引和一个“B 树”索引，而且就算它们都基于同一个数据列也没问题。

有几种类型的比较操作不能借助索引来完成。如果只通过把数据列值传递到 STRCMP()等函数的办法去进行比较，就享受不到任何索引带来的好处。服务器必须为每个数据行计算出函数值，这就排除了使用该数据列上的索引的可能性。

利用“慢查询”日志找出性能低劣的查询。这个日志可以帮助我们找出有可能受益于使用索引的查询命令。（对 MySQL 各种日志的讨论见 12.5 节）“慢查询”日志是一个文本文件，可以用任何一种文件阅读程序去查看，也可以用 mysqldumpslow 工具程序来汇总它的内容。如果在这个日志里经常看到某个查询命令，就应该想到它的代码可能不够优化。应该尝试改写它以加快它的运行速度。在查看“慢查询”日志的时候要记住：“慢”是实时测量出来的。因此，如果把一个负载沉重的服务器和一个负载较轻的服务器相比，在前者的“慢查询”日志里会出现更多的查询命令。

5.2 MySQL 的查询优化程序

当你发出一个选取数据行的查询语句时，MySQL 就会分析它，并考虑是否可以对它进行优化以加快查询。在本节中，我们将讨论查询优化程序是如何工作的。要想获得更多的资料，请参阅《MySQL 参考手册》中关于优化的一章，其中讨论了 MySQL 所采用的各种优化方法。

当然，MySQL 的查询优化程序要充分地利用索引，但它也同时使用其他的信息。例如，假定你发布了下面的查询，不管这个数据表有多大，MySQL 都会十分迅速地执行它。

```
SELECT * FROM tbl_name WHERE FALSE;
```

在这个例子中，MySQL 查看 WHERE 子句，并且意识到没有数据行可以符合查询的条件，因此根本就没有再去搜索数据表。你可以发出一个 EXPLAIN 语句来看到这一点，这个语句可以要求 MySQL 显示出一些信息，表明如何在没有实际执行 SELECT 的情况下执行了这个查询。要使用 EXPLAIN 语句，把单词 EXPLAIN 放在 SELECT 语句前面就可以了：

```
mysql> EXPLAIN SELECT * FROM tbl_name WHERE FALSE\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: NULL
        Extra: Impossible WHERE
```

通常情况下，EXPLAIN 语句会返回较多的信息，包括将要用于扫描数据表的索引（除 NULL 外）的信息，将要用到的联结的类型，以及需要从每个数据表上扫描的数据行数目的大概估计。

在某些场合，EXPLAIN 会实际执行给定查询命令的某部分。比如说，如果查询命令的 FROM 子句里包含子查询，EXPLAIN 必须先执行子查询才能知道它们将返回些什么，然后才能对主 SELECT 语句进行分析。

5.2.1　查询优化器的工作原理

查询优化程序有好几个目标，主要目的是只要可能就要使用索引，并且要使用条件最严格的索引来尽可能多、尽可能快地排除那些不符合索引条件的数据行。后一部分听上去与常理相悖，因为它和我们的直觉不符。毕竟，你发出 SELECT 语句的目标是找到数据行，而不是排除数据行。查询优化程序之所以这样工作是因为它从搜索范围中排除数据行的速度越快，找到那些与搜索条件匹配的数据行也就越快。假如首先针对最严格的条件进行测试检验，查询就能够进行得更快一些。假定你有一个查询，它要检验两个数据列，每一个都有一个索引：

```
SELECT col3 FROM mytable
WHERE col1 = 'some value' AND col2 = 'some other value';
```

同时假定检验 col1 时有 900 个匹配的数据行，检验 col2 时有 300 个匹配的数据行，同时检验两个数据列时有 30 个匹配的数据行。如果首先检验 col1，就要从 900 个数据行中找到其中与 col2 匹配的 30 个，也就是说有 870 个数据行经过检验是不符合查询条件的。如果首先检验 col2，则要 300 个数据行中找到其中与 col1 匹配的 30 个，也就是说仅有 270 个数据行经过检验是不符合查询条件的，因此它只需要较少的计算和磁盘输入/输出。由此可见，查询优化程序会首先检验 col2，因为这样做的工作量较小。

下面给出的指导意见可以帮助优化器充分利用索引。

对数据表进行分析。这将生成关于索引值分布情况的统计数据，它们可以帮助优化器对索引的使用效果做出更准确的评估。在默认的情况下，当你把一个有索引的数据列里的值与一个常数相比较的

时候，优化器会假设相关索引里的键值是均匀分布的。在判断是否应该把某个索引用于常数比较操作的时候，优化器还会对该索引进行一次快速检查以估算需要用到多少个索引项。对于 MyISAM 和 InnoDB 数据表，可以主动地使用 ANALYSE TABLE 语句让服务器对键值进行一次分析。

如果某个数据表在填充好数据之后就不再发生变化，只需在加载后对它做一次分析就够了。如果某个数据表经常更新，就应该时不时地（这要根据数据更新的频繁程度来决定）对它进行分析。

使用 EXPLAIN 语句来验证优化器操作。EXPLAIN 语句可以告诉你某给定查询有没有使用索引。如果你正在尝试使用不同的办法来编写同一条语句，或者如果你想知道增加某种索引能否改善查询命令的执行效率，这类信息可以给你很大的帮助。这方面的例子见 5.2.2 节。

向优化器提供提示或在必要时屏蔽之。在联结操作中，你可以在数据表列表中的某个数据表名字的后面利用 FORCE INDEX、USE INDEX 或 IGNORE INDEX 限定词告诉服务器你想使用哪些索引。详见附录 E 中对 SELECT 语句的描述。

还可以利用 STRAIGHT_JOIN 强制优化器按特定的顺序使用数据表。在正常情况下，MySQL 优化器会自行确定按照何种顺序扫描数据表才能最快地把数据行检索出来。但在少数场合，优化器做出的决定并不一定是最优的。如果遇到这样的事情，你可以用 STRAIGHT_JOIN 关键字“覆盖”优化器的选择。使用了 STRAIGHT_JOIN 的联结操作类似于一个交叉关联，但将迫使数据表按照它们在 FROM 子句里出现的先后顺序相互关联。

如果你真想这么做，就应该在列出数据表时把那个将被选取的数据行个数最少的数据表放在第一个。要是对它是哪个数据表没有把握，那就把数据行个数最多的那个数据表放在第一个好了。这里的原则是，安排数据表的顺序是为了让限制性最强的选取操作最先执行。候选数据行的个数缩减得越快、越早，查询的执行性能就越高。

STRAIGHT_JOIN 可以在 SELECT 语句里的两个地方使用。其一是放在 SELECT 与输出数据列清单之间，这将使它对语句里的所有交叉联结起作用；其二是放 FROM 子句里。下面两条语句是等效的：

```
SELECT STRAIGHT_JOIN ... FROM t1 INNER JOIN t2 INNER JOIN t3 ... ;
SELECT ... FROM t1 STRAIGHT_JOIN t2 STRAIGHT_JOIN t3 ... ;
```

应该在使用和未使用 STRAIGHT_JOIN 的情况下分别测试一下查询的执行情况。MySQL 也许会有很好的理由不按照你认为的最好顺序去使用索引，STRAIGHT_JOIN 有可能起不到帮助作用。（用 EXPLAIN 语句检查一下执行计划，看 MySQL 是如何处理每一条语句的。）

尽量使用数据类型相同的数据列进行比较。在对带有索引的数据列进行比较时，如果它们的数据类型相同，查询性能就会高一些；如果它们的数据类型不同，查询性能就会低一些。比如说，INT 不同于 BIGINT，所以进行一次 INT/INT 或 BIGINT/BIGINT 比较就要比进行一次 INT/BIGINT 比较的速度更快。例如，CHAR(10)就被认为和 CHAR(12)或者 VARCHAR(10)是相同类型的数据列，但实际上与 CHAR(12)和 VARCHAR(12)不同。如果你要比较的数据列是不同类型，你可以使用 ALTER TABLE 语句来修改其中一个数据列，从而使它们类型相同。

使带索引的数据列在比较表达式中单独出现。如果你在函数调用中使用了一个数据列，或者使将数据列作为算术表达式中很复杂的项目中的一部分，MySQL 将不能使用索引，因为它必须要对每一个数据行计算出表达式的值。虽然有时这是无法避免的，但在许多情况下你都可以重新写出查询来使带索引的数据列单独出现。

下面的两个 WHERE 子句说明了如何处理这种情况。从算术角度来讲，它们是完全一样的，但从优化方面看可就大不一样了。

```
WHERE mycol * 2 < 4
WHERE mycol < 4 / 2
```

对第一个 WHERE 子句来说，MySQL 必须检索数据列 mycol 中每一个数据行的值，然后乘以 2，再将所得到的结果和 4 比较。在这种情况下，不能使用索引，因为数据列中的每一个值都要被检索以计算出比较中左侧的表达式的值。对第二个 WHERE 子句来说，优化程序将表达式 4/2 简化为数值 2，然后对数据列 mycol 使用索引，很快就能找到所有小于 2 的数值。可见，第二条语句比第一条好。

让我们来看另外一个例子。假定你有一个带索引的数据列 date_col。如果你发出了一个如下所示的查询，这个索引是不能使用的：

```
SELECT * FROM mytbl WHERE YEAR(date_col) < 1990;
```

这个表达式没有将带索引的数据列和 1990 比较；它比较的是一个从数据列中计算出的一个值，而且必须要对每一个数据行都进行计算才能得到这个值。结果，数据列 date_col 的索引就不能使用，因为完成这个查询需要的是对数据表的全部扫描。怎样来解决这个问题呢？只要使用一个准确的日期表示方法，数据列 date_col 的索引就可以使用，从而找到数据列中与之匹配的值：

```
WHERE date_col < '1990-01-01'
```

但是，假定你又没有一个准确的日期，你或许只是想找到哪些数据行是从今天开始往前的某一天。有几种方式可以表达这种类型的比较，效果不尽相同。3 种可能的方式如下所示：

```
WHERE TO_DAYS(date_col) - TO_DAYS(CURDATE()) < cutoff
WHERE TO_DAYS(date_col) < cutoff + TO_DAYS(CURDATE())
WHERE date_col < DATE_ADD(CURDATE(), INTERVAL cutoff DAY)
```

对第一种方式，不能够使用索引，因为数据列的每个数据行都要被检索才能计算出 TO_DAYS (date_col)的值。第二种方式要好一些，cutoff 和 TO_DAYS(CURDATE( ))都是常数，所以比较表达式右边的值可以在处理查询之前就由优化程序一次计算出来，而不用逐行计算。但是数据列 date_col 仍然会出现在函数调用中，因此，索引还是不能使用。第三种方式最好。同样地，表达式右边的数值能够在查询执行之前计算出一个常数，但是现在的值是一个日期。这个日期值可以直接和数据列 date_col 中的值比较，不需要再转换成天数。在这种情况下，可以使用索引。

不要在 LIKE 模式的开始位置使用通配符。有些时候，人们使用下面格式的 WHERE 子句来对字符串进行搜索：

```
WHERE col_name LIKE '%string%'
```

如果要找到所有出现在数据列中的字符串，那么这个子句是正确的。但不要出于习惯而将符号“%”放在字符串的两侧。如果只想找到出现在数据列开始位置的字符串，就需要删除第一个“%”。假定你要在数据列中查找以“Mac”开始包含姓氏的姓名，如 MacGreGor 和 MacDougall，就需要像下面这样写出 WHERE 子句：

```
WHERE last_name LIKE 'Mac%'
```

优化程序看到了模式的字符开始部分，会使用索引来找到匹配的数据行，就好像你写出的是下面这样的表达式，这种格式的表达式允许对 last_name 使用索引:

```
WHERE last_name >= 'Mac' AND last_name < 'Mad'
```

这种优化的方法不能用于使用 REGEXP 操作符的模式匹配。REGEXP 表达式不能被优化。

利用优化器的长处。MySQL 支持联结和子查询，但子查询支持是从 MySQL 4.1 版开始才增加的功能。因此，在许多场合，优化器对联结的优化效果要比对子查询的优化效果更好一些。这在某个子查询运行得很慢时有很现实的指导意义。正如 2.9.7 节里讨论的，有些子查询可以改写为效果相同的联结。如果慢速子查询是其中之一，就应该尝试把它改写为一个联结，看是否运行得更快。

试验各种查询的变化格式，而且要多次运行它们。当试验一个查询的变化格式（例如子查询和等效联结）时，每一种方式都要多运行几次。假如你对一个查询的两种不同方式都只运行一次，你会经常发现第二种查询方式要快一些，实际上这只不过是因为第一次查询的信息仍然保留在磁盘的高速缓存内，不再需要从磁盘读取。还要注意，应该在系统负载相对稳定时运行查询，这样可以避免系统上的其他活动对试验结果产生影响。

避免过多使用 MySQL 的自动类型转换功能。MySQL 能够实现自动类型转换，但是如果能避免这些转换，你或许能得到更好的性能。例如，如果数据列 num_col 是一个整数数据列，下面的两个查询都将返回同样的结果：

```
SELECT * FROM mytbl WHERE num_col = 4;
SELECT * FROM mytbl WHERE num_col = '4';
```

但是第二个查询包含类型转换。这个转换操作对整数和字符串进行了重复转变来完成比较，因而本身在性能方面要受到一个小小的损失。比较严重的问题是，如果数据列 num_col 有索引，那么，含有类型转换的比较有可能会阻止索引得到使用。

与之相对的比较类型（把一个字符串数据列与一个数值相比较）也会阻止索引的使用。假设编写了一个如下所示的查询命令：

```
SELECT * FROM mytbl WHERE str_col = 4;
```

具体到这个例子，str_col 数据列上的索引将无法使用，因为在 str_col 数据列里可能会有许多不同的字符串值在转换为一个数值后等于 4（例如'4'、'4.0'、'4th'等），要想把它们都找出来，唯一的办法就是依次读取、转换，再进行比较。如果想寻找的是某个特定的字符串值，例如'4'，为了避免上述问题，在查询命令里应该像下面这样给出：

```
SELECT * FROM mytbl WHERE str_col = '4';
```

5.2.2 用EXPLAIN语句检查优化器操作

EXPLAIN 语句提供的信息可以帮助我们了解优化器为处理各种语句而生成的执行计划。在这节里，我将演示 EXPLAIN 语句的两种用途。

- 了解以不同方式编写出来的查询命令是否会影响到索引的使用。
- 了解给数据表增加索引对优化器生成高效执行计划的能力会产生什么影响。

本节将只描述与具体示例有关的那些 EXPLAIN 输出字段，对 EXPLAIN 输出的详细讨论见附录 E。书中给出的输出内容是我在我的系统上看到的结果。根据你的服务器版本和具体配置，你看到的结果也许会有所不同。

5.2.1 节讨论过，编写一个表达式的方式可以决定优化器能否使用现成的索引。具体地说，那时候的讨论中给出了 3 个效果相同的 WHERE 子句作为例子，只有第三个允许使用索引：

```
WHERE TO_DAYS(date_col) - TO_DAYS(CURDATE()) < cutoff
WHERE TO_DAYS(date_col) < cutoff + TO_DAYS(CURDATE())
WHERE date_col < DATE_ADD(CURDATE(), INTERVAL cutoff DAY)
```

EXPLAIN语句可以让我们了解以某种方式编写表达式是否优于另一种。为了让大家有一个直观的印象，让我们一起使用每个WHERE子句来搜索一下member数据表里的expiration数据列的值，在搜索时把 cutoff 的值统一设置为 30 天。请注意，在刚被创建出来的时候，member 数据表在expiration 数据列上没有任何索引。要想看到索引的使用与表达式编写方式之间的关系，必须先为expiration数据列创建一个索引：

```
mysql> ALTER TABLE member ADD INDEX (expiration);
```

接下来，用EXPLAIN语句依次检查那几个表达式，看优化器为它们生成的执行计划是什么样的：

```
mysql> EXPLAIN SELECT * FROM MEMBER
    -> WHERE TO_DAYS(expiration) - TO_DAYS(CURDATE()) < 30\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: MEMBER
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 102
        Extra: Using where
mysql> EXPLAIN SELECT * FROM MEMBER
    -> WHERE TO_DAYS(expiration) < 30 + TO_DAYS(CURDATE())\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: MEMBER
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 102
        Extra: Using where
mysql> EXPLAIN SELECT * FROM MEMBER
    -> WHERE expiration < DATE_ADD(CURDATE(), INTERVAL 30 DAY)\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: MEMBER
         type: range
possible_keys: expiration
          key: expiration
      key_len: 4
          ref: NULL
         rows: 6
        Extra: Using where
```

从前两条语句的EXPLAIN输出结果可以看出，它们都没有使用索引。在EXPLAIN的输出结果里，type字段的值将告诉我们数据是如何从数据表读出的。ALL的意思是“将检查所有的数据行”，也就是将进行一次不借助于任何索引的全表扫描。名字里有key字样的各个字段的值都是NULL同样表明

将不使用任何索引。

作为对照，第三条语句的 EXPLAIN 输出结果表明，像那样编写的 WHERE 子句会让优化器使用 expiration 数据列上的索引。

- ❑ type 字段的值表明，优化器可以使用索引来搜索一个特定区间内的值（小于表达式的右半部分所给出的天数的那些值）。
- ❑ possible_key 和 key 字段的值表明，expiration 数据列上的索引被视为一个候选索引，并最终会被实际使用。
- ❑ row 字段的值表明，优化器估计它需要检查 6 个数据行才能完成这次查询，这比前两个执行计划里的 102 个数据行好多了。

EXPLAIN 语句的第二个用途是验证一下增加索引能否帮助优化器更有效地执行一条语句。具体到下面这个例子，我将只使用两个数据表，它们一开始都没有任何索引。这已足以演示创建索引的效果，而这里得出的结论将同样适用于涉及更多数据表的更复杂的联结操作。

假设我们有两个名为 t1 和 t2 的数据表，两个数据表各有 1 000 个数据行，包含从 1 到 1 000 的值。下面是我们将检验的查询命令，它将根据条件从这两个数据表生成一个结果集：

```
mysql> SELECT t1.i1, t2.i2 FROM t1 INNER JOIN t2
    -> WHERE t1.i1 = t2.i2;
+------+------+
| i1   | i2   |
+------+------+
|    1 |    1 |
|    2 |    2 |
|    3 |    3 |
|    4 |    4 |
|    5 |    5 |
...
```

在这两个数据表都没有任何索引的情况下，EXPLAIN 语句的输出结果如下所示：

```
mysql> EXPLAIN SELECT t1.i1, t2.i2 FROM t1 INNER JOIN t2
    -> WHERE t1.i1 = t2.i2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
```

```
          ref: NULL
         rows: 1000
        Extra: Using where
```

请看，type 字段的值 ALL 表明将进行一次需要检查所有数据行的全表扫描。possible_keys 字段的值 NULL 表明没有找到可以加快查询速度的候选索引。(由于没有合适的索引，key、key_len 和 ref 字段的值也都是 NULL。)Using where 表明将使用 WHERE 子句里的信息去寻找符合条件的数据行。

以上信息告诉我们，优化器没有发现任何可以帮助这条查询命令执行得更有效率的有用信息，它将采取以下行动。

- 对 t1 数据表进行一次全表扫描。
- 为 t1 数据表里的每个数据行对 t2 数据表进行一次全表扫描，使用 WHERE 子句里的信息去寻找符合条件的数据行。

rows 字段的值是优化器对它在查询过程的每个阶段需要检查多少个数据行的预估值。因为需要进行一次全表扫描，所以 t2 数据表的这个预估值是 1 000。类似地，t2 数据表的这个预估值也是 1 000，但对 t2 数据表的全表扫描需要为 t1 数据表里的*每一个数据行*进行一遍。换句话说，按照优化器的估计，在处理这个查询命令时需要检查的数据行组合的个数是 1 000×1 000，也就是 100 万。这也太浪费了，因为总共只有 1 000 种组合可以满足 WHERE 子句里的条件。

为了让查询更有效率，给其中一个相关联的数据列增加一个索引，然后再次运行 EXPLAIN 语句：

```
mysql> ALTER TABLE t2 ADD INDEX (i2);
mysql> EXPLAIN SELECT t1.i1, t2.i2 FROM t1 INNER JOIN t2
    -> WHERE t1.i1 = t2.i2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1000
        Extra:
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
         type: ref
possible_keys: i2
          key: i2
      key_len: 5
          ref: sampdb.t1.i1
         rows: 10
        Extra: Using where; Using index
```

情况有了改善。t1 数据表的 EXPLAIN 结果没有变化（对它仍将进行一次全表扫描），但优化器对 t2 数据表的处理不同了。

- type 字段的值从 ALL 变成了 ref，意思是可以通过使用一个引用值（来自 t1 数据表的值）进行一次索引搜索来定位 t2 数据表里符合条件的数据行。

- ❑ 该引用值在 ref 字段里给出：sampdb.t1.i1。
- ❑ row 字段的值从 1 000 下降到了 10，这表明优化器认为只需要为 t1 数据表里的每个数据行检查 10 个 t2 数据表里的数据行就够了。（这是一个悲观的估计。事实上，在 t2 数据表里只能为 t1 数据表里的每个数据行找到一个与之匹配的数据行。稍后将会看到如何帮助优化器做出更准确的估计。）数据行组合的总预估值是 $1\,000 \times 10 = 10\,000$。这比此前没有任何索引时的预估值 100 万可少多了。

有必要给 t1 数据表也加上索引吗？具体到这个联结操作，无论如何也需要对其中一个数据表进行全表扫描，而进行全表扫描是不需要任何索引的。为了看出它到底有没有影响，下面给 t1.i1 数据列也加上一个索引，然后再次运行 EXPLAIN 语句：

```
mysql> ALTER TABLE t1 ADD INDEX (i1);
mysql> EXPLAIN SELECT t1.i1, t2.i2 FROM t1 INNER JOIN t2
    -> WHERE t1.i1 = t2.i2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: index
possible_keys: i1
          key: i1
      key_len: 5
          ref: NULL
         rows: 1000
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
         type: ref
possible_keys: i2
          key: i2
      key_len: 5
          ref: sampdb.t1.i1
         rows: 10
        Extra: Using where; Using index
```

如上所示的 EXPLAIN 结果和前一条 EXPLAIN 语句的很相似，但新增加的索引让 t1 数据表的 EXPLAIN 输出有了些变化：type 字段的值从 NULL 变成了 index，Extra 字段的值从空白变成了 Using index。这些变化表明，虽然仍需要对已经有了索引的 t1 数据表进行一次全表扫描，但优化器现在可以直接从 t1 数据表的索引读到 t1.i1 数据列的值，用不着读它的数据文件了。对于一个 MyISAM 数据表，当优化器知道从该数据表的索引文件就可以获得它所需要的全部信息时，它就会给出这样的 EXPLAIN 结果。对于一个 InnoDB 数据表，当优化器只使用来自索引的信息就可以完成查询任务而无需搜索以获得数据行时，它也将给出这样的 EXPLAIN 结果。

接下来，还可以采取一个步骤以帮助优化器对查询开销做出更准确的预估：运行 ANALYSE TABLE 命令。这将使服务器生成关于键值分布情况的统计数据。分析 t1 和 t2 数据表并再次运行 EXPLAIN 语句，就可以在 rows 字段里看到更准确的预估值了：

```
mysql> ANALYZE TABLE t1, t2;
```

```
mysql> EXPLAIN SELECT t1.i1, t2.i2 FROM t1 INNER JOIN t2
    -> WHERE t1.i1 = t2.i2\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: t1
         type: index
possible_keys: i1
          key: i1
      key_len: 5
          ref: NULL
         rows: 1000
        Extra: Using index
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: t2
         type: ref
possible_keys: i2
          key: i2
      key_len: 5
          ref: sampdb.t1.i1
         rows: 1
        Extra: Using where; Using index
```

具体到这个例子，优化器最新的估计是 t1 数据表里的每一个值将只与 t2 数据表里的一个数据行相匹配。

5.3　为提高查询效率而挑选数据类型

对数据类型的选择可以从几个方面影响查询性能。本节提供了一些关于如何挑选数据类型的建议，它们可以帮助查询命令运行得更快。

尽量使用数值操作，少使用字符串操作。数值运算通常要比字符串运算快得多。以比较操作为例，数值之间的比较只用一个操作就可以完成；而字符串之间的比较一般需要进行多次字节与字节或字符与字符的比较才能完成，而且字符串越长，比较的次数就越多。

如果字符串数据列的可取值的个数是有限的，选用 ENUM 或 SET 类型可以在数值操作中受益。这些类型在 MySQL 内部是以数值形式表示的，所以处理效率要比其他字符串类型更高一些。

考虑使用另一种字符串表示方法。有时候，把字符串表示为数字可以显著改善性能。例如，以点记号给出的 IP 地址（如 192.168.0.4）通常用一个字符串来表示。但这种 IP 地址很容易转换为整数形式：把构成 IP 地址的 4 组数字依次存入 INT UNSIGNED 类型的 4 个字节之一。整数不仅节约存储空间，还可以加快检索速度。但从另一方面看，把 IP 地址表示为 INT 值会给模式匹配操作带来困难，而模式匹配操作是在寻找给定子网里的 IP 地址时肯定会用到的。也许你可以用位掩码操作来“模拟”模式匹配操作，但这会让你的代码变得相当复杂。这类问题告诫我们：不能只能考虑空间问题，你必须根据数据的具体用途和用法为它们挑选一种最适当的表示形式。（具体到 IP 地址这个例子，不管你的选择是什么，INET_ATON()和 INET_NTOA()函数都可以帮你完成 IP 地址在整数和字符串两种表示形式之间的转换。）

如果“小”类型够用，就不要选用“大”类型。“小”类型要比“大”类型的处理速度更快。尤

其是字符串，它们的处理用时与它们的长度呈正比关系。选用“小”类型的另一个好处是可以让整个数据表变得更小，从而减少在磁盘读写方面的开销。对于那些有索引的数据列，使用较短的数据值还将进一步提升其整体性能。这一方面是因为索引本身就可以加快查询的速度，另一方面则是因为短索引值的处理速度比长索引值的处理速度更快。

对于使用固定长度数据类型的数据列，应该根据其取值范围选用最小的类型。如果 MEDIUMINT 类型够用，就不要选用 BIGINT 类型；如果 FLOAT 类型足以满足精度要求，就不要选用 DOUBLE 类型。如果你决定选用固定长度的 CHAR 数据列，就不要把它们设置得太长。如果某个 CHAR 数据列里最长的值是 40 个字符长，就不要把它定义为 CHAR(255)，CHAR(40)已经足够了。

对于可变长度的类型，选用较小的类型仍可以节约空间。BLOB 类型使用 2 个字节来记录值的长度，而 LONGBLOB 类型要使用 4 个字节。如果你将要保存的数据没有长度大于 64KB 的，选用 BLOB 可以在每个值上替你节约 2 个字节。（类似的考虑也适用于 TEXT 类型。）

如果你能选择数据行的存储格式，就应该尽量选用最适用于你的存储引擎的格式。对于 MyISAM 数据表，就应该尽量选用固定长度的数据列而不是可变长度的数据列。具体地说，就是尽量使用 CHAR 而不是 VARCHAR 数据列来保存字符串数据。这样做的缺点是你的数据表将占用较多的空间，但如果你能负担得起额外的空间，固定长度的数据行将比可变长度的数据行处理得更快。那些经常需要修改、因而更容易形成碎片的数据表更是如此，如下所示。

- 对于可变长度的数据行，因为数据行的长度不一致，所以在经过许多删除或更新操作后将形成更多的碎片。你将需要定期运行 OPTIMIZE TABLE 语句以保持性能。固定长度的数据行就没有这样的问题。
- 如果数据表发生崩溃，固定长度的数据行更容易重新构造。因为数据行的长度是固定的，所以每个数据行的开头都出现在数据行长度的整数倍位置上，很容易确定，而为可变长度的数据行确定开头位置就没有这么容易了。这是一个与查询处理没有关系的性能问题，但它肯定可以加快数据表的修复过程。

虽然把 MyISAM 数据表转换为使用固定长度的数据行可以改善性能，但你应该首先考虑以下因素。

- 固定长度的数据列虽然速度快，但会占用更多的空间。CHAR(*n*)数据列里的每个值都要占用 *n* 个字符的空间（即使是空白值也是如此），MySQL 在插入较短的值时会用尾缀空格把它补足到数据列的宽度。VARCHAR(*n*)数据列占用的空间比较少，因为 MySQL 只为每个值分配容纳它所必需的字符空间，再加上 1 个或 2 个字节来保存它的长度。因此，选用 CHAR 数据列还是 VARCHAR 数据列归根结底是一个你想节约时间还是想节约空间的问题。如果速度是你最关心的问题，选用 CHAR 数据列将使你获得固定长度数据列的性能优势。如果空间紧缺，VARCHAR 数据列应该是你的首选。作为一般原则，尽管固定长度的数据行会占用比较多的空间，但仍可以改善性能。但对一个非常关键的应用程序而言，你应该用两种办法分别实现数据表并进行一些测试以确定哪一种更适合你的具体应用。
- 有时候，就算你想使用固定长度的类型，也没有办法做到。比如说，目前还没有一种固定长度的类型能够容纳长度超过 255 个字符的字符串。

MEMORY 数据表目前使用固定长度的格式来存储数据行，所以选用 CHAR 还是 VARCHAR 都无关紧要，它们都将被当做 CHAR 类型来对待。

对于 InnoDB 数据表，因为它的数据行内部存储格式对固定长度的数据列和可变长度的数据列不

加区分（每个数据行有一个标头，存放着指向关数据列的指针），所以使用固定长度的 CHAR 数据列不见得比使用可变长度的 VARCHAR 好。因此，影响性能的主要因素是数据行所占用的存储量。事实上，因为 CHAR 类型通常要比 VARCHAR 类型占用更多的空间，所以从减少空间占用量和减少磁盘 I/O 操作次数的角度讲，使用 VARCHAR 类型反而更有利。

尽量把数据列声明为 NOT NULL。如果数据列具有 NOT NULL 属性，对它的处理可以更快地完成，因为 MySQL 不再需要在查询处理期间检查该数据列的值是不是 NULL。它还可以为每个数据行节约一个位的存储空间。在数据列里避免使用 NULL 值可以让你的查询命令变得更简单，因为你不再需要把 NULL 值当做一个特例来检查，而更简单的查询命令通常可以更快地得到处理。

考虑使用 ENUM 数据列。如果字符串数据列的不同取值的个数是有限的，就应该把它转换为 ENUM 数据列。ENUM 值在 MySQL 数据库内部被表示为一系列数值，所以它们的处理速度很快。

利用 PROCEDURE ANALYSE()语句。你可以利用 PROCEDURE ANALYSE()语句来分析数据表，看它会对数据列的声明提出哪些建议：

```
SELECT * FROM tbl_name PROCEDURE ANALYSE();
SELECT * FROM tbl_name PROCEDURE ANALYSE(16,256);
```

输出里有一个输出列，其内容是对数据表里的各数据列的优化建议。第二条 PROCEDURE ANALYSE()语句的含义是，如果数据列的不同取值在 16 个以上或者长度超过了 256 个字节（你可以根据具体情况改变这两个数字），就不提出使用 ENUM 类型的建议。如果不加上这个限制条件，输出可能会非常长；ENUM 类型的声明通常很难阅读。

根据 PROCEDURE ANALYSE()语句的输出，可以把数据列修改为一种更优化的类型了。如果决定要改变数据列的类型，可以使用 ALTER TABLE 语句来进行。

对容易产生碎片的数据表进行整理。数据表——尤其是那些包含可变长度数据列的数据表——往往会因为数据的大量修改而产生碎片。碎片可不是什么好东西，它会在你用来存放数据表的硬盘上造成空洞（未使用的空间）。随着时间的推移，你将不得不读取更多的存储块才能把你想要的数据行读入内存，这无疑会对性能产生不良的影响。包含可变长度数据行的数据表都会产生碎片，BLOB 或 TEXT 数据列受到的影响往往是最大的，因为它们的长度变化是如此之大。

定期使用 OPTIMIZE TABLE 语句有助于防止数据表查询性能的降低。OPTIMIZE TABLE 语句可以用来清理 MyISAM 数据表里的碎片。对各种存储引擎都适用的碎片整理办法是这样的：先用 mysqldump 工具程序转储数据表，再利用转储文件删除并重建一个，如下所示：

```
% mysqldump db_name tbl_name > dump.sql
% mysql db_name < dump.sql
```

把数据压缩到 BLOB 或 TEXT 数据列里。在应用程序里，用 BLOB 或 TEXT 数据列来保存数据（你可以对这些数据进行压缩和解压缩）能让你只用一个检索操作就把你需要的东西都找出来。这个办法特别适合用来存储那些很难用标准的数据表结构来表示的数据和那些会随时间而变化的数据。在第 2 章介绍 ALTER TABLE 语句时，我们给出了一个用来充当 Web 题库的数据表例子。在那个例子里，当需要往题库里增加新考题时，我们会使用 ALTER TABLE 语句给那个数据表增加一个数据列。

这类问题的另一种解决方案是让负责处理 Web 表单的应用程序把考题数据压缩到某种数据结构里，然后再插入到一个 BLOB 或 TEXT 数据列里。比如说，你可以把用户给出的考题答案表示为 XML 字符串，再把这个 XML 字符串保存到一个 TEXT 数据列里。这将给应用程序的客户端增加一些负担（对数据编码，以及从数据表里检索数据行后再解码），但大大简化了数据表的结构——在需要修改题库

时，你用不着改变数据表的结构。

事情总是一分为二的。BLOB 和 TEXT 值也会给它们自己带来一些麻烦，尤其是在你进行了大量的 DELETE 或 UPDATE 操作时。删除这些值会在数据表里留下一个大空洞，这个空洞将由一个或多个不同长度的数据行来陆续填补。（你可以用刚刚介绍的碎片处理来对付这个问题。）

使用人造索引。这种“人造索引”数据列往往会有一种出奇制胜的效果。具体做法之一是先根据数据表里的其他数据列计算出一个散列值并把它另外保存到一个数据列里，然后通过搜索散列值去检索你想找的数据行。注意：这个技巧只适用于精确匹配型查询。（散列值在“<”或“>=”等操作符进行的范围搜索中毫无用处。）散列值可以用 MD5()函数来生成。SHA1()或 CRC32()函数也可以用来生成散列值。当然，你也可以在应用程序里用你自己的算法来计算你自己的散列值。请记住，数值类型的散列值的存储效率是非常高的。还有一点要特别注意：如果你选用的散列算法有可能生成带有尾缀空格的字符串值，就不要选用那些会自动去除尾缀空格的数据类型来存储它们。

人造散列索引对 BLOB 或 TEXT 数据列非常有用。与直接搜索 BLOB 或 TEXT 数据列本身的做法相比，通过散列索引检索 BLOB 或 TEXT 值要快得多。

尽量避免对很大的 BLOB 或 TEXT 值进行检索（除非万不得已）。比如说，如果不能断定 WHERE 子句将把查询结果限制为你想要找的那些数据行，贸然使用一个 SELECT *查询检索所有数据行就不可取。你很可能会让数据库服务器把一些大的 BLOB 或 TEXT 值通过网络传输给你。这是人造索引——BLOB 或 TEXT 值的散列标识信息——大显身手的又一领域。你应该先搜索数据列以确定哪些数据行符合筛选条件，然后再从符合条件的数据行里把你想要的 BLOB 或 TEXT 值检索出来。

把 BLOB 或 TEXT 数据列剥离到单独一个数据表里。在某些场合，把数据表里的 BLOB 或 TEXT 数据列剥离出来并存放到另一个数据表（如果这有助于把数据表里的其他数据列组织为固定长度的数据行格式）往往会是一个不错的主意。这既能减少原始数据表里的碎片，使你享受到使用固定长度的数据行所带来的好处；又能使原始数据表上的 SELECT *查询不会把大的 BLOB 或 TEXT 值通过网络传输给你。

5.4 有效加载数据

在大部分时间里，你最关心的问题大概是对 SELECT 查询的优化，因为它们是最常用的查询类型，也因为要明白如何优化它们并不总是那么简单。相比而言，将数据加载到数据库里则简单明了多了。你可以采用一些方法来提高数据加载的效率。基本的原则如下所述。

- 批量加载的效率比单数据行加载的效率高，因为键缓存在每一次输入的记录加载以后都不需要刷新，它可以在批量记录结束的时候再刷新。越是减少键缓存的刷新次数，数据加载也就越快。（对索引的修改是在键缓存区里进行的，然后才会在适当时机写到硬盘上。与将该缓存刷新、许多次相比，只刷新一次显然可以减少磁盘 I/O 操作。）
- 加载有索引的数据表比加载比加载无索引的数据表快一些。如果有一些索引，不但数据行要添加到数据表里，每一个索引也必须得到修改来反映出所添加的新行。
- 较短的 SQL 语句的数据加载较长的语句快，因为它们在服务器中包含较少的语法分析，同时也因为它们能够更快地从客户程序经过网络送到服务器。

这些因素有些看上去是微不足道的（特别是最后一个因素），但是如果加载大量的数据，即使很小的影响效率的因素也会形成不同的结果。根据上面讨论的一些一般原则，我们可以就如何快速地加载数据得出几个实用的结论。

使用LOAD DATA（各种格式）语句要比使用INSERT语句效率高，因为它批量加载数据行。服务器只需要对一个语句（而不是几个语句）进行语法分析和解释。索引只在所有数据行都处理完后才需要刷新，而不是每处理一行都刷新。

使用LOAD DATA语句要比使用LOAD DATA LOCAL语句效率高。使用LOAD DATA语句，文件必须要位于服务器上，而且你必须要有FILE权限，但是服务器可以直接从磁盘上读取文件。使用LOAD DATA LOCAL语句，客户程序要先读取文件，然后再通过网络将它送到服务器上，这种方式就要慢一些了。

如果你只能使用INSERT语句，那就要使用将多个数据行在一个语句中给出的格式：

```
INSERT INTO tbl_name VALUES(...),(...),... ;
```

你在语句中给出的数据行越多，效果越好。这将会减少你需要的语句总数，最大程度地减少了索引刷新的次数。这看上去和我们前面谈到的较短的语句比较长的语句处理得更快的结论相矛盾。但实际上是不矛盾的。这个原则指的是一条插入多个数据行的INSERT语句要比每行插入一个数据行的多条INSERT语句短一些，而且多数据行语句在服务器上的处理需要的索引刷新少得多。

如果用mysqldump工具程序来生成数据库备份文件，它会默认生成含有多个数据行的INSERT语句。你也可以使用--opt（优化）选项，它自动地打开--extended-insert选项，生成含有多个数据行的INSERT语句，以及其他一些在重新加载时使转储文件得到高效处理的选项。

要避免使用mysqldump工具程序的--complete-insert选项，否则，产生的INSERT语句将用于单个的数据行，而语句较长，比多数据行语句需要更多的语法分析。

如果你只能使用多个INSERT语句，要尽可能地对它们分组以减少索引的刷新次数。对于事物性存储引擎，要在一个事务内发出INSERT语句而不用自动提交的方式来实现这一点：

```
START TRANSACTION;
INSERT INTO tbl_name ... ;
INSERT INTO tbl_name ... ;
INSERT INTO tbl_name ... ;
COMMIT;
```

对于非事务性的存储引擎，获得对数据表的写锁定，在数据表被锁定时，发出INSERT语句。

```
LOCK TABLES tbl_name WRITE;
INSERT INTO tbl_name ... ;
INSERT INTO tbl_name ... ;
INSERT INTO tbl_name ... ;
UNLOCK TABLES;
```

这样一来，在上面的两种情况下你都获得了同样的益处。索引只在所有语句都执行完以后才刷新，而不是对每一个INSERT语句都要刷新一次，如果是用自动提交方式或是数据表没有被锁定，就会出现这样的问题。

对于MyISAM数据表，减少索引刷新次数的另一个策略是使用DELAY_KEY_WRITE数据表选项。在使用了这个选项之后，数据行仍将像往常一样被立刻写入数据文件，但键缓存将只在必要时才刷新一次而不是每次插入一个新索引值后就立刻刷新一次。要想在服务器全局范围内获得上述DELAY_KEY_WRITE效果，必须在启动mysqld程序时给出--delay-key-write=ALL选项。如此启动mysqld程序之后，数据表的索引块写操作将被推迟到发生以下事件时才进行：必须刷新某个索引块以便为其他的索引值腾出空间的时候，执行了FLUSH TABLES语句的时候，数据表被关闭的时候。

请注意，如果为MyISAM数据表启用了“键延迟写”功能，服务器意外关机事件将有可能导致

索引值的丢失。但因为 MyISAM 数据表的索引可以根据它的数据行得到修复，所以这并不是个致命问题。不过，为了保证修复工作确实会发生，应该在启动服务器时给出`--myisam-recover=FORCE`选项。这个选项将强制服务器在打开 MyISAM 数据表时对它们进行检查并自动地进行必要的修复。

在复制机制中的从服务器上，可以用`--delay-key-write=ALL`选项为所有的 MyISAM 数据表统一启用“键延迟写”功能，用不着顾虑它们在主服务器上是如何创建的。

使用压缩的客户/服务器协议来减少通过网络的数据量。对大部分的 MySQL 客户程序，可以使用`--compress` 命令行选项来指定这一点。通常情况下，这种做法只用于速度较低的网络上，因为压缩过程要占用一部分处理器时间。

让 MySQL 为你插入默认值。也就是说，不要在 `INSERT` 语句中定义数据列，不管怎样它都会被设置为默认值。平均起来，你的语句要短一些，减少了经由网络送到服务器的字符数目。此外，因为这些语句包含较少的值，服务器只需进行较少的语法分析和值转换。

对 MyISAM 数据表来说，如果你需要将大量的数据加载到一个新的数据表里，创建没有索引的数据表要快一些，先加载数据，然后再创建索引。一次创建全部索引要比对每行修改它们快一些。对于一个已经有索引的数据表来说，如果你先撤销或使索引失效，然后再重建或是重新使索引生效，数据加载有可能会快一些。这些方法不适用于 InnoDB 数据表，它对分开的索引创建没有优化方法。

如果你正在考虑使用先撤销或使索引失效的方法来将数据加载到 MyISAM 数据表里，要全面考虑你当时的情况，正确地评估你可以获得哪些益处。如果你是要将少量的数据加载到一个大的数据表中去，重新构建索引的方法可能要比不作特殊准备就加载数据的方法需要更长的时间。

要撤销并重新构建索引，使用 `DROP INDEX` 和 `CREATE INDEX` 语句或 `ALTER TABLE` 语句有关索引的格式。要使索引失效和重新生效，有两种选择：

- 你可以使用 `ALTER TABLE` 语句的 `DISABLE KEYS` 和 `ENABLE KEYS` 格式：

  ```
  ALTER TABLE tbl_name DISABLE KEYS;
  ALTER TABLE tbl_name ENABLE KEYS;
  ```

  这些语句对数据表中的所有非唯一索引进行关闭和打开。`ALTER TABLE` 的 `DISABLE KEYS` 和 `ENABLE KEYS` 语句在使索引失效和重新生效的操作中应该是首选的方法，因为使用它们时服务器并不工作。（注意，如果使用 `LOAD DATA` 语句来将数据加载到一个空白的 MyISAM 数据表中，服务器会自动完成对它的优化。）

- `myisamchk` 工具程序能够完成索引操作，它直接操作数据表文件，因此，要想使用它们，必须具有对数据表写入访问的权限。你还应该注意 14.2 节中要讨论的一些措施，来使服务器在你使用数据表文件时不能访问这个数据表。

  要想通过 `myisamchk` 使 MyISAM 数据表索引失效，首先要确定已经通知了服务器，使这个数据表单独存放，然后再将它移到合适的数据库目录中，并且运行下面的命令：

  ```
  % myisamchk --keys-used=0 tbl_name
  ```

  将数据加载到数据表以后，重新使索引生效：

  ```
  % myisamchk --recover --quick --keys-used=n tbl_name
  ```

  n 作为位掩码来指示要使哪一个索引生效。掩码 0（第一位）对应着索引 1。例如，假如一个数据表有 3 个索引，*n* 的值就应该是 7（二进制的 111）。你可以使用`--description`选项来决定索引的编号：

```
% myisamchk --description tbl_name
```

前面所讨论的数据加载的原则也适用于包含着完成不同种类操作的客户程序的混合查询环境。例如，你通常希望避免对那些经常被改变（写入）的数据表长时间运行 SELECT 查询。这样会引发写入者的许多争用问题以及较差的性能。如果你的写入大部分是 INSERT 操作，解决这个问题的一种可能方式是，先将数据行添加到一个辅助的数据表中，然后再将这些数据行定期地添加到主数据表中。如果你需要能够迅速地访问新数据行，就不能使用这种方法了，但是如果你可以在短时间内不访问它们，那么使用辅助数据表会在以下两个方面对你提供帮助。首先，在 SELECT 查询发生在主数据表上时，这种方法减少了争用的情况，因此它们可以执行得更快一些。其次，从辅助数据表中将数据行批量加载到主数据表中要比单独加载数据行省时，键缓存只是在每一批（而不是每个）数据行加载完成后被刷新。

当你在记录从 Web 服务器的 Web 页面访问 MySQL 数据库时，就可以使用这个方法。在这种情况下，要确定立刻进入到主数据表的项目不一定是要优先考虑的事项。

如果在某个 MyISAM 数据表上混合出现 INSERT 和 SELECT 操作，可以考虑使用“并发插入”（concurrent insert）功能来提高总体效率。这个功能可以让插入操作与检索操作同时进行而无需使用辅助数据表。详见 5.5.3 节。

5.5　调度和锁定问题

前面几节集中讨论了怎样加快各个查询的速度。你有权改变语句的调度优先权，这样就使得好几个客户程序的查询可以更协调地工作，从而使各个客户程序不致于被长时间地锁在外面。改变优先权的做法也可以保证特殊种类的查询可以被处理得更快一些。本节讨论 MySQL 的默认调度策略以及你可以用于改变这个策略的选项。本节还会讨论共发插入的作用和存储引擎的锁定级别对多个客户程序并发的影响。为了方便讨论，把完成检索（SELECT）的客户程序作为读取者，把修改数据表（DELETE、INSERT、REPLACE 或 UPDATE）的客户程序作为写入者。

MySQL 的默认调度策略综述如下。

- 写入比读取有更高的优先权。
- 对数据表的写操作必须按照“写”请求先来后到的顺序一个接一个地进行。
- 对同一个数据表进行的读操作可以同时进行。

对 MyISAM、MERGE 和 MEMORY 存储引擎来说，调度策略是由数据表锁帮助完成的。只要客户程序访问数据表，首先必须锁定它。当这个客户程序完成了对数据表的操作时，对它的锁定才能够解除。用 LOCK TABLES 和 UNLOCK TABLE 语句来显式获得和解除锁定是可以的，但是通常情况下，服务器的锁定管理程序能够自动地在需要时获得锁定，在不需要时解除它们。所需要的锁定类型取决于客户程序是在写入还是在读取。

对数据表进行写入操作的客户程序必须具有对数据表唯一访问的锁定。在操作进行过程中，数据表处于不稳定的状态，因为数据行一直被删除、添加或改变，数据表上的所有索引可能会随之更新。当数据表处在不断变化的状态时，允许其他客户程序访问它会引起问题。允许两个客户程序同时对一个数据表进行写入操作，显然不是一件好事，因为这会很快将数据表毁坏成不能使用的混乱状态。允许一个客户程序从一个不稳定的数据表中读取数据也不是一件好事，因为数据表被读取时可能正发生着变化，读取结果是不准确的。

对数据表进行读取的客户程序必须锁定它，防止其他客户程序对数据表进行写入操作，或者在你读取数据表时修改它。但是，这种锁定不需要具备对于读取操作的唯一访问，其他的客户程序可以同时从这个数据表上读取数据。读取数据并不改变数据表，因此，没有理由禁止其他读取者访问这个数据表。

MySQL 提供了一些语句修饰符来改变它的调度策略。一是在 DELETE、INSERT、LOAD DATA、REPLACE 和 UPDATE 语句中使用 LOW_PRIORITY 关键字，二是在 SELECT 和 INSERT 语句中使用 HIGH_PRIORITY 关键字，三是在 INSERT 和 REPLACE 语句中使用 DELAYED 关键字。

LOW_PRIORITY 和 HIGH_PRIORITY 限定符只对支持数据表锁定功能的存储引擎（MyISAM、MERGE 和 MEMORY）有效果。DELAYED 限定符在 MyISAM、MEMORY、ARCHIVE 和（从 MySQL 5.1.19 版开始）BLACKHOLE 存储引擎上都可以使用。

5.5.1 改变语句的执行优先级

LOW_PRIORITY 关键字影响 DELETE、INSERT、LOAD_DATA、REPLACE 和 UPDATE 语句的调度。通常情况下，假如一个数据表正在被读取时来了一个写入操作，写入者就会受到阻止直到读取者完成读取工作。（因为查询一旦开始，它就不会被干扰，所以读取者被允许完成读取。）如果在写入者等待时，另外一个读取请求又来了，那么这个读取者也被锁定，因为默认的调度策略是写入者比读取者优先。当第一个读取者完成以后，写入者开始工作，写入者完成任务以后，第二个读取者再开始工作。

假如写入请求是一个 LOW_PRIORITY 的请求，那么写入就没有比读取更高的优先级。在这种情况下，假如在写入者等待时有第二个读取请求到来，那么第二个读取者允许插队到写入者的前面。只有等到不再有读取者到来时，写入者才能够开始工作。从理论上来说，这种调度方式意味着 LOW_PRIORITY 的写入请求有可能会一直被锁定。当前一个读取操作正在进行时，只要有其他的读取请求到来，它们就会插到 LOW_PRIORITY 的写入请求之前。

对于具有 HIGH_PRIORITY 关键字的 SELECT 查询，情况也是类似的。它允许 SELECT 查询操作插入到正在等待的写入操作之前，即使通常情况下写入操作有较高的优先。另一个影响是高优先级的 SELECT 语句将先于普通的 SELECT 语句执行，这是因为普通的 SELECT 语句会被写操作阻塞，要等写操作完成后才能执行。

如果想把所有支持 LOW_PRIORITY 选项的语句都默认当做一条低优先级语句来对待的话，就需要在启动服务器时给出--low-priority-updates 选项。此后，如果需要把一条 INSERT 语句提升到正常的写优先级，只要写一条 INSERT HIGH_PRIORITY 语句就可以临时取消该选项的效果。

5.5.2 使用延迟插入

DELAYED 修饰符用在 INSERT 和 REPLACE 语句中。当 DELAYED 请求到达数据表时，服务器将数据行排序并且迅速地返回到客户程序的状态，从而使客户程序可以在数据行被插入之前进行。假如读取者正在从数据表读取数据，队列中的数据行将保持状态直到没有读取者。然后服务器开始在延迟的数据行队列中插入数据行。从这时开始，服务器定期检查是否有新的读取请求到来并且在等待着，如果有，这个延迟的数据行队列就被挂起来，同时允许读取者开始工作。当所有读取者的工作完成后，服务器再一次开始插入延迟的数据行。这个过程持续到队列为空。

LOW_PRIORITY 和 DELAYED 修饰符在某种意义上有些类似，它们都允许数据行的插入延迟，但是

在对客户程序操作的影响方式上却有很大的不同。LOW_PRIORITY 是迫使客户程序等待数据行能够被插入,而 DELAYED 是使客户程序继续进行,并且服务器可以在内存中缓冲数据行直到有时间处理它们。

如果其他客户程序正在运行着很长的 SELECT 语句，而你又不想一直锁在那里等待插入完成，这时 INSERT DELAYED 就很有用处。具有 INSERT DELAYED 的客户程序可能进行得快一些，因为服务器会使要插入的数据行排队。

但是，你应该意识到一般的 INSERT 和 INSERT DELAYED 操作之间的其他不同之处。如果 INSERT DELAYED 语句中包含语法错误，客户程序会返回错误提示，但是却没有返回通常情况下你可能得到的一些信息。例如，当语句返回时，你不能指望会得到 AUTO_INCREMENT 的值。你也不会得到对唯一索引进行复制的数目。之所以会出现这样的情况，是因为在操作实际完成之前插入操作就返回了客户程序的状态。这还意味着，INSERT DELAYED 语句中的数据行在内存中排队时，如果服务器出现崩溃或是用 kill -9 来删除，这些数据行就不存在了。普通的 kill -TERM 删除不会这样（服务器在退出之前插入了数据行)。

5.5.3　使用并发插入

MyISAM 存储引擎对读取者锁定写入者的通用原则有一个特例。这种情况是发生在 MyISAM 数据表在数据文件里没有空洞的时候（例如当删除或更新数据行时)，在这种情况下，INSERT 语句只能在数据行的末尾而不是中间位置添加数据行。这时，客户程序可以在其他客户程序正在从数据表中读取数据时向数据表中添加数据行。因为这些操作可以在检索没有被锁定时进行，因而被称之为“并发插入”。使用并发插入时，要注意下面的问题。

- 不要在 INSERT 语句中使用 LOW_PRIORITY 修饰符。这会使 INSERT 语句总是锁定读取者从而导致并发插入不能完成。
- 那些需要显式锁定数据表但又想允许并发插入的读取者应该使用 LOCK TABLES...READ LOCAL 而不是 LOCK TABLES...READ。关键字 LOCAL 会使你得到一个允许并发插入的锁，因为它只用于在数据表中已经存在的数据行，并不锁定正被添加到数据表末尾的新数据行。
- 应该给 LOAD DATA 操作加上 CONCURRENT 限定符，让给定数据表上的 SELECT 语句在该数据表正在加载数据的同时仍可以执行。
- 内部有空洞的 MyISAM 数据表不能用于并发插入。不过，你可以用 OPTIMIZE TABLE 语句对有空洞的数据表进行碎片整理。这将消除那些空洞，至少在又发生过删除或更新操作之前会是如此。

5.5.4　锁定级别与并发性

上面讨论的调度修饰符可用于影响默认的调度策略。在很大程度上，它们最初用来处理数据表级别的锁定使用中出现的问题，这就是 MyISAM、MERGE 和 MEMORY 存储引擎用来处理数据表竞争的方法。

InnoDB 存储引擎可以在不同级别上完成锁定，因此在竞争的管理方面具有不同的性能特征。InnoDB 使用数据行级别的锁定，但是只在需要的时候才这样做。(在许多情况下，比如只有读取操作在进行时，InnoDB 数据表可能根本就没有锁定。)

存储引擎所使用的锁定级别在客户程序之间出现的并发问题上有明显的作用。假定有两个客户程

序，都想在给定的数据表里修改数据行。要完成这个修改，每个客户程序都需要一个写锁定。对于 MyISAM 类型的数据表来说，引擎会使第一个客户程序拥有数据表锁，将第二个客户程序锁定，直到第一个客户程序完成操作。对于 InnoDB 类型的数据表，并发问题更加突出：只要修改的不是同一个数据行，修改操作就可以同时进行。

一般的原则是，较小范围的数据表锁定会出现较好的并发性，因为如果各自使用数据表的不同部分，就有更多的客户程序可以同时使用这个数据表。也就是说，不同的存储引擎可以较好地适应不同的查询混合。

- MyISAM 类型的数据表检索时非常快。但是，使用数据表级别的锁定对混合检索和修改可能不利，特别是在检索可能要运行较长时间时。在这种情况下，在修改能够进行之前，可能需要长时间的等待。
- 在有较多修改操作时，InnoDB 类型的数据表可以提供较好的性能。因为锁定只是数据行级，而不是数据表级，数据表被锁定的范围比较小。这样就可以减少竞争问题，提高并发性。

从死锁的防止措施来说，数据表锁定比小范围锁定好。使用数据表锁定，死锁问题绝不会出现。服务器会决定哪些数据表需要对查询锁定，并会提前将它们全部锁定。对于 InnoDB 类型的数据表来说，死锁问题可能会出现，因为在事务开始的时候，处理程序没有获得所有必要的锁定。在事务处理的过程中，只在必要时，才需要锁定。因此，有可能出现这种情况，两个查询会获得锁定，然后会在已经保持的锁定被解除时试图获得进一步的锁定。结果，每个客户程序在它能继续进行之前保持着其他客户程序需要的锁定。这就导致了死锁问题的产生，服务器必须中止其中的一个事务处理。

5.6 系统管理员所完成的优化

前面各节讨论了没有特殊权限的 MySQL 用户所能够完成的各种优化。还有一些优化只能由管控着 MySQL 服务器或是管控着 MySQL 运行的机器的管理员才能完成。例如，一些服务器参数与查询过程有关，而且可以调整。同时，某些硬件配置因素对查询速度有直接的作用。在许多场合，这些优化措施将改善服务器的整体性能，因而对全体 MySQL 用户都有好处。

总体来说，在完成管理优化的时候，需要记住的主要原则如下所述。

- 在内存中访问数据比从磁盘上访问数据快。
- 在内存中尽可能长地保存数据可以减少磁盘活动量。
- 保留索引的信息要比保留数据行的内容更加重要。

应用这些原则最常见的方法就是增加服务器缓存的容量。服务器有许多你可以改变从而影响其运行的参数（系统变量），其中有几个直接影响查询过程的速度。你可以改变的最重要的参数是数据表缓存容量的大小以及数据表处理程序缓存索引操作信息所使用的缓存。将可用内存分配到服务器的缓存将会使信息在内存中保存得更长，从而减少磁盘活动量。这样的做法是好的，因为从内存中获取信息要比从磁盘上读取信息快许多。

- 为了让文件打开操作的次数最小化，服务器在打开数据表文件之后会尽量让它们继续保持在打开状态。具体地说，它会把已经打开的文件的信息保存在数据表缓存里。`table_cache` 系统变量（在 MySQL 5.1 版里是 `table_open_cache` 系统变量）控制着这个缓存的大小。如果服务器访问过大量的数据表，数据表缓存将逐渐被填满，服务器就会关闭一些最近没有使用过的数据表，为将要打开的新数据表腾出空间。如果你想了解数据表缓存的工作情况，请查看 `Opened_tables` 状态变量：

```
SHOW STATUS LIKE 'Opened_tables';
```

Opened_tables 指示出一个数据表曾经被打开多少次后就没有被打开过。（这个值在 mysqladmin status 命令的输出中也作为 Opens 值显示。）如果这个数保持稳定或者缓慢增加，就说明设置可能是正确的。如果这个数增加很快，就意味着缓存已满，有些数据表必须被关闭从而为打开新的数据表准备空间。如果你有文件描述符，增加数据表缓存大小会减少打开数据表的次数。

- MyISAM 键缓冲区用来保持 MyISAM 数据表有关索引的操作的索引块。它的大小由服务器变量 key_buffer_size 控制。较大的值允许 MySQL 可以同时在内存中保持较多的索引块，这就增加了在内存中找到键值（而不用从磁盘上读取新的数据块）的可能性。默认的键缓冲区是 8 MB。如果你有大量内存，这个默认值就过于保守了，应该可以根据实际情况增加它，将看到基于索引的检索和对索引的创建和修改操作都会有明显的性能改善。

  你可以为 MyISAM 数据表创建额外的键缓存并分配给特定的数据表使用。这对那些数据表的查询处理工作会很有帮助。参见 5.6.1 节里的解释。
- InnoDB 存储引擎有它自己用于缓冲数据和索引值的缓存，其大小由 innodb_buffer_pool_size 系统变量控制。InnoDB 存储引擎还维护着一个日志缓冲区，其大小由 innodb_log_buffer_size 变量控制。
- 另外一个特殊的缓存是查询缓存，5.6.2 节将对它进行讨论。

在 12.6.1 节中可以找到系统变量设置的说明。你在改变参数值时，要遵循下列原则。

- 一次只改变一个参数。否则，你就是在改变多个独立的变量，这会使得对参数的改变进行评估非常困难。
- 逐步增加系统变量值。如果你认为变量越大越好，因而将某个变量值一次增加很大，这样的话，你可能耗光资源，导致系统运行缓慢或瘫痪，因为你将变量值设得太高了。
- 在 MySQL 服务器上试验和调整各种参数的风险很大，谨慎的做法是另外设置一个测试服务器。
- 要想得知哪种变量参数适用于你的系统，可以看看包含在 MySQL 发行版本中的 my-small.cnf、my-medium.cnf、my-large.cnf 以及 my-huge.cnf 选项文件。（在 Unix 下，你可以在资源发行版本中的 support_files 目录下或二进制发行版本中的 share 目录下找到它们。在 Windows 下，它们位于基本安装目录中，文件名后缀是.ini.。）
- 这些文件不仅可以让你了解都有哪些参数最需要随着服务器的使用情况的不同而改变，还可以让你熟悉那些参数的典型设置值。

还有一些可以使服务器高效率运行的方法，如下所述。

- 禁用那些你不需要的存储引擎。服务器不会给被禁用的存储引擎分配任何内存，因此你可以将内存用于他处。如果是从源代码开始自行编译 MySQL 软件，在编译配置阶段就可以把用不着的存储引擎排除在服务器的可执行代码以外。即使是那些必须被收录在服务器里的存储引擎，有许多也可以在启动服务器时通过适当的启动选项加以禁用。这方面的细节见 12.7.1 节。
- 让权限表尽可能简单。虽说服务器会把权限表的内容缓存在内存里，但只要 tables_priv、column_priv 或 procs_priv 数据表里有数据行，服务器就必须在检查 SQL 语句的权限时去检查它们的内容。如果那些数据表是空的，服务器就可以优化它的权限检查流程，跳过那些权限级别。
- 如果你是在源点构建 MySQL 服务器，要用静态库而不是共享库来配置它。使用共享库的动态

二进制在磁盘空间上保存，但静态二进制要快得多。但是，如果你想使用用户定义的功能（UDF）的机制，就不能使用静态二进制，因为系统需要动态链接。在这种系统中，静态二进制不起作用。

5.6.1 使用MyISAM键缓存

如果 MySQL 执行的一条语句使用了来自 MyISAM 数据表的索引，它将使用一个键缓存来容纳那些索引值。这种缓存有助于减少磁盘 I/O 操作：如果能在键缓存里找到来自某个数据表的键值，就用不着从磁盘上把它们再次读取出来了。可是，键缓存毕竟只是一个容量有限的资源，而且在默认的情况下还是由所有的 MyISAM 数据表所共享的。如果没能在缓存里找到想要的值并且缓存已经满了，就将导致竞争。因此，必须从该缓存丢弃一些现有的值以便为新值腾出空间，下次再需要用到刚才丢弃的那些值时，又不得不把它们再次从磁盘读回来。

如果你有一个使用频率非常高的 MyISAM 数据表，设法让它的键长期驻留在内存里是有好处的，但刚才提到的竞争现象会让你很难保证这一点。导致竞争的原因既有可能是需要从同一个数据表读取键，也有可能是需要从其他数据表读取键。如果把键缓存加大到能够把某给定数据表的所有键全部容纳在其中，就可以避免来自同一个数据表的键发生竞争，但来自其他数据表的键仍会竞争这个缓存里的空间。

MySQL 已经为这个问题准备了一个解决方案：MySQL 支持建立多个键缓存，我们可以为某个数据表分配一个键缓存并把该数据表的索引提前加载到该缓存。这很有用，如果你有一个使用频率非常高的数据表和足够多的内存，就应该把该数据表的索引全部加载到一个足够大的键缓存里。这种能力使我们既可以避免来自同一个数据表的键发生竞争，也可以避免来自其他数据表的键竞争这个缓存里的空间：创建一个足够大的键缓存来容纳某个数据表的所有索引，并把该缓存设定为仅供该数据表使用。在把键加载到缓存里之后，就不再需要进行磁盘 I/O 操作了。当然，也不再需要从缓存丢弃键值，与这个数据表有关的键检索操作在内存里就可以全部完成。

下面的例子演示了如何为 sampdb 数据库里的 member 数据表创建一个键缓存，这个缓存的名字是 member_cache，长度是 1MB。必须具备 SUPER 权限才能执行下面给出的命令：

(1) 创建一个新的键缓存，让它大到足以容纳来自 member 数据表的索引：

```
mysql> SET GLOBAL member_cache.key_buffer_size = 1024*1024;
```

(2) 把 member 数据表指定给这个键缓存：

```
mysql> CACHE INDEX member IN member_cache;
+---------------+--------------------+----------+----------+
| Table         | Op                 | Msg_type | Msg_text |
+---------------+--------------------+----------+----------+
| sampdb.member | assign_to_keycache | status   | OK       |
+---------------+--------------------+----------+----------+
```

(3) 把 member 数据表的索引提前加载到它的键缓存里去：

```
mysql> LOAD INDEX INTO CACHE member;
+---------------+--------------+----------+----------+
| Table         | Op           | Msg_type | Msg_text |
+---------------+--------------+----------+----------+
| sampdb.member | preload_keys | status   | OK       |
+---------------+--------------+----------+----------+
```

如果你想把其他数据表加载到同一个缓存或者是为其他的数据表创建其他的键缓存，套用以上步骤即可。如果想了解关于键缓存的更多信息，请参阅 12.7.2 节。

用上述步骤和语句创建的专用键缓存在服务器重新启动之后将不复存在。如果想在服务器每次启动之后都可以使用这种缓存，就必须安排这些语句在服务器每次重启时都能得到执行。比如说，可以把它们放到一个文件里并用`--init-file`服务器选项来命名那个文件。

5.6.2　使用查询缓存

MySQL 服务器可以使用查询缓存来加快重复执行的 `SELECT` 语句的过程。这样做导致的性能改善经常是十分明显的。查询缓存有以下特点。

- 一个给定的 `SELECT` 语句第一次执行时，服务器记下了它查询的文本和返回的结果。
- 服务器下一次看到这个查询时就不再执行它，而是直接从查询缓存中将查询结果取出并返回给客户程序。
- 查询缓存以服务器接受的那些查询字符串的文字文本为基础。如果查询文本是完全一样的，查询就被看成是相同的。如果字母大小写上不同，或者来自于使用着不同字符集或通信协议的客户程序，查询就被认为是不同的。如果查询是完全一样的，但实际上指的根本就不是相同的数据表，那么查询也会被认为是不同的。（例如，假如它们指的是不同数据库中名称相同的数据表。）
- 如果某个查询命令返回的结果不确定，这个查询就不会被缓存。比如说，一个使用了 `NOW()` 函数的查询会随着时间的变化而返回不同的结果，所以它不能被缓存。
- 当一个数据表被更新时，所有与之相关的缓存着的查询会全部失效，并被删除。这样能防止服务器返回已经被更改了的旧结果。

对查询缓存的支持被构建成默认设置。如果你不想使用这个缓存，想避免遭受它所包含的哪怕是最小的额外开销，你可以在运行 `configure` 脚本时使用`--without-query-cache` 选项来构建不带查询缓存的服务器。

要想知道某个服务器是否支持查询缓存，查看一下 `have_query_cache` 系统变量的值即可：

```
mysql> SHOW VARIABLES LIKE 'have_query_cache';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| have_query_cache | YES   |
+------------------+-------+
```

模　式	含　义
0	不缓存查询结果/不检索已被缓存的结果
1	缓存查询，但不包括以SELECT SQL_NO_CACHE开头的查询
2	只缓存以SELECT SQL_CACHE开头的查询

对于支持查询缓存的服务器来说，缓存操作会受到以下 3 个系统变量值的影响。

- `query_cache_type` 决定查询缓存的操作模式。
- `query_cache_size` 决定了为查询缓存分配的内存大小，以字节为单位。下面显示了可能的模式值。

- query_cache_limit 设置能够缓存的最大结果集的大小，比这个值大的查询结果不能被缓存。

例如，为了使查询存储能用，为它分配 16 MB 的内存，要在选项文件中使用下列设置：

```
[mysqld]
query_cache_type=1
query_cache_size=16M
```

即使 query_cache_type 为 0，query_cache_size 指定的内存量也会分配到。为避免浪费内存，将大小设置为 0，除非要启用缓存。注意，即使 query_cache_type 为非零值，为 0 的大小也会禁用缓存。

每个客户程序都以服务器默认的缓存方式所指示的状态开始查询缓存行为。使用下面的语句，客户程序可以改变查询的默认缓存方式：

```
SET query_cache_type = val;
```

val 可以是 0、1 或 2，这几个值的含义相当于在启动服务器时把全局级 query_cache_type 系统变量设置为同样的值。在 SET 语句里，允许使用符号值 OFF、ON 和 DEMAND，它们分别对应于数值 0、1 和 2。

在 SELECT 关键字后面添加一个修饰符，客户程序也可以控制各个查询的缓存。如果缓存模式是 ON 或 DEMAND，那么 SELECT SQL_CACHE 将使可缓存查询的结果被缓存。SELECT SQL_NO_CACHE 使结果不被缓存。

对于那些从不断变化的数据表中检索信息的查询来说，禁止缓存是比较有用的。在这种情况下，缓存没有太多的用处。假设你把 Web 服务器的请求记录在了一个 MySQL 的数据表里，并且你还要周期性地运行数据表上的一系列摘要查询。由于 Web 服务器非常繁忙，经常新的数据行插入到数据表里，因此，数据表被缓存的查询结果会很快失效。这就意味着尽管你可以重复地发出摘要查询，但是查询缓存对于这种重复起不了什么作用。在这种情况下，明智的做法是，使用 SQL_NO_CACHE 修饰符来发出查询通知服务器没有必要缓存它们的结果。

5.6.3 硬件优化

本章前面讨论了许多帮助你改善服务器性能的技术，但没有考虑到硬件配置。你当然可以利用较好的硬件来使你的服务器运行得更快。但并不是所有硬件方面的改变都有相同的价值。在评估你需要进行哪些方面的硬件改进时，最重要的原则与调整服务器参数时的原则一样。将尽可能多的信息快速储存起来，并且尽可能久地保存它们。

你可以对硬件配置的以下几个方面进行修改来改进服务器性能。

在机器里安装更多的内存。这会增加服务器的缓存和缓冲区的容量，从而允许数据在内存中能够保持更长的时间，较少需要从磁盘上获取信息。

如果你有足够的 RAM 可以让所有的数据交换都发生一个内存文件系统里，可以重新配置你的系统来删除所有的磁盘数据交换设备。要知道，即使你有足够的 RAM，有些系统仍会与磁盘交换数据。

添加更快的磁盘来改善 I/O 等待时间。在这里，寻道时间通常是性能的主要决定因素。侧向移动磁头是比较慢的。当磁头位置确定以后，把信息从磁道上读出来相对要快一些。不过，要是你能在更多的内存和更快的磁盘之间做选择，应选更多的内存。内存永远比磁盘快，添加内存后可以使用较大的缓存并减少磁盘的活动量。

在物理设备之间分散磁盘读写活动，提高并行度。通过多个物理设备将读和写分开，比在一个设备上读和写要快。例如，假设将数据库存储在一个设备上，将日志文件存储在另外一个设备上，那么对两个设备同时写入就要比数据库和日志文件都在同一个设备上时快一些。注意，在同一个物理设备上采用不同分区是没有用的，不能算并行，因为它们还是要竞争同一个物理资源（磁头）。移动日志文件和数据库的方法请见11.3节。

在把数据重新分配到不同的设备之前，一定要知道系统的加载特性。如果在某个物理设备上已经发生了一些其他的主要活动，那么把数据库移到那里会使性能变差。例如，假设你处理许多的Web业务，但又将数据库移到了Web服务器文本树所在的设备上，那就不会有任何的性能改进。

RAID设备的使用也可以给你一些并行的益处。

使用多处理器硬件。对于MySQL服务器这样的多线程程序来说，多处理器硬件可以同时执行多个线程。

Part 2

第二部分

MySQL 编程接口

本部分内容

第 6 章

MySQL 程序设计

对于撇开 MySQL 发行版本所收录的标准化客户程序不用而自行编写基于 MySQL 的程序的一些理由，本章进行了探讨。还对随后几章将要讨论的 3 种程序设计语言（C、Perl 和 PHP）的 MySQL 编程接口做了概念性介绍，并讨论了在挑选程序设计语言时需要考虑的因素。

6.1　为什么要自己编写 MySQL 程序

MySQL 发行版本收录了一系列客户程序。比如说，mysqldump 程序可以导出数据表的定义和内容，mysqlimport 程序可以把数据文件加载到数据表里，mysqldamin 程序可以执行管理操作，mysql 程序可以让你与服务器通信以执行任意的 SQL 语句。

那些标准的客户端程序足以解决 MySQL 用户日常遇到的绝大多数问题，但有些应用需求超出了它们的能力范围。为解决这类问题，MySQL 服务器特意准备了一个客户端 API（Application Programming Interface，应用程序编程接口），我们可以利用该接口提供的灵活性满足应用程序可能会有的任何特殊要求。该客户端 API 的基本功能是访问 MySQL 服务器，它可以用来干什么就要看你的想象力了。

在本书的这个部分，我们将对在编写和使用基于 MySQL 的程序去访问数据库时需要掌握的知识展开讨论。为了让大家了解自行编写程序可以带来什么样的收获，下面把如此可以完成的事情与对 mysql 程序的能力及其提供的与 MySQL 服务器的简单接口的使用做个对比。

- **可以对输入处理环节进行定制**。在使用 mysql 程序的时候，只能输入原始的 SQL 语句；而在自行编写的程序里，可以向用户提供更直观和更容易使用的输入方法。只要编写的程序足够好，就可以让它的用户不必掌握 SQL 语言，甚至觉察不出数据库在他们正在完成的任务里扮演的角色。数据输入界面可以简单到只有一个用来提示用户并读入数据的命令行接口，也可以复杂到是用某种屏幕管理工具包（如 curses 或 S-Lang）实现的全屏数据录入表单、用 Tcl/Tk 实现的一个 X 窗口、Web 页面里的一个表单，等等。

  对大多数人来说，通过填写一个表单的方式来给出搜索参数要比发出一条 SELECT 语句容易得多。比如说，一位正在根据价格范围、建筑形式或地点寻找房子的房地产经纪人，肯定只想以最省事的法子把搜索参数输入一个表单并获得符合条件的房源清单。输入新数据行或更新现有数据行时也是如此：数据录入人员只需要知道在行内输入哪些值，而不需要掌握 INSERT、REPLACE 或 UPDATE 等语句的 SQL 语法。
- **可以对用户提供的输入进行检查**。比如说，可以检查他们输入的日期是否符合 MySQL 所预期的格式，也可以要求必须填写某些字段。这可以提高应用程序的安全性。

- **可以自动生成输入数据**。有些应用可能根本不需要有人的参与，比如当 MySQL 需要的输入数据是由其他程序自动生成的时候。你也许会配置 Web 服务器把日志记录项写入一个 MySQL 数据库而不是一个文件，也许会设置定期运行的系统监控程序并且把系统状态信息写入一个数据库，等等。
- **可以对输出进行定制**。mysql 程序的输出内容没有什么格式，只能选择使用制表符来分隔数据项或使用最简单的表格样式。如果想获得更美观的输出，就必须自己动手去排版它。这方面的要求可以简单到只是把 NULL 值替换为“无”，也可以复杂到按规定生成一份报告。请看下面这份报告：

```
State  City         Sales
------------------------------
AZ     Mesa          $94,384.24
       Phoenix       $17,328.28
       -----------------------
       Subtotal     $117,712.52
------------------------------
CA     Los Angeles $118,198.18
       Oakland       $38,838.36
       -----------------------
       Subtotal     $157,036.54
==============================
       TOTAL        $274,749.06
```

  这份报告有以下几个值得注意的地方。
  - 定制的标题。
  - State 列没有重复的值，只在它们发生变化时才显示出来。
  - 增加了 Sutotal（小计）和 TOTAL（总计）计算。
  - 把 94384.24 这样的数值排版为美元金额格式$94,384.24。

  另一项常见的复杂格式化任务是制作发票：把发票抬头、顾客信息和商品信息按规定格式排好。这类输出报告的复杂程度很容易超出 mysql 程序的格式化能力。

  对于某些任务，你或许不想让它生成任何输出。比如说，检索信息是为了进行计算，而计算结果应该立刻存入另一个数据表。或者，你想把输出发送到其他地方而不是显示给运行这个查询命令的用户。比如说，你想检索出来的姓名和电子邮件地址自动发往另一个进程以批量生成一些通知函。程序确实会生成一些输出，但其内容却是给收件人而不是运行这个程序的那个人看的。
- **可以克服 SQL 语言本身的某些先天不足**。绝大多数 SQL 脚本由一条条语句构成，它们将从头到尾依次执行，中间只进行最少的出错检查。如果用 mysql 程序的批处理模式去执行一个 SQL 查询文件，mysql 程序要么会在遇到第一个错误时退出，要么（如果你给出了--force 选项的话）一口气执行完所有的查询而不管中间发生了多少个错误。如果是你自己编写的程序，就可以有选择地在查询命令的前后加上一些流程控制语句，对其执行成功或是失败的情况做相应的处理。可以根据某个查询命令的成功或失败去执行另一个查询命令，也可以根据前一个查询命令的结果去决定下一步该做什么。

  我们知道，MySQL 支持使用存储过程，该机制提供的流程控制结构和出错处理结构给 SQL 语言增加了一些灵活性，但那些结构所提供的灵活性是无法与各种通用程序设计语言相提并

论的。

SQL 语句之间只有非常有限的联系，把某个查询命令结果用作另一个查询命令的输入或是把多个查询命令的结果组织到一起都很困难。虽然可以用 LAST_INSERT_ID()函数来获得前面的语句最近一次生成的 AUTO_INCREMENT 值，也可以对用户变量赋值并在后面的语句里引用它们，但也仅此而已。这种局限性使人们很难单独使用 SQL 语言来实现一些非常有用的操作，例如，遍历检索出来的数据行以完成一系列比较复杂的处理。如果打算检索出一群顾客，再详细查询他们每个人的信用记录，整个过程可能需要为每一位顾客执行多个查询。

总之，涉及“主从”关系和有着复杂输出格式要求的任务都不适合使用 mysql 程序来完成，应该考虑使用其他的工具。这里的基本思路是：利用能够提供“胶水”的程序，把查询命令“粘”在一起，让我们能够把某个查询命令的输出用作另一个查询命令的输入。

- **可以把 MySQL 集成到任何应用程序里去**。有很多种程序会受益于数据库的信息提供能力，而 MySQL 客户端应用程序接口可以让它们获得这种能力。有了这种能力，当应用程序需要核对顾客号码或查看库存情况时，只要发出一条查询命令即可；当 Web 应用程序收到某个客户查询某位作者所有作品的请求时，可以从一个数据库把它们检索出来并把检索结果发送回那个客户的浏览器。

  应该承认，用 mysql 命令执行一个由 SQL 语句构成的输入文件，再用其他工具对其输出做进一步处理，这样利用 shell 脚本多少也可以获得一些把 MySQL“集成”到应用程序里的效果。但这种做法失于灵活，不适用于情况比较复杂的场合。如果一直采用这种“集成”方案，这种脚本会越来越多，其调试和补丁工作会越来越麻烦，到最后往往会陷入一种“不用没办法，用起来很麻烦”的尴尬局面。不仅如此，利用 shell 脚本执行其他命令的做法在进程创建方面的开销很大，偶尔使用一次没什么，但长此以往恐怕就承受不起了。更有效率的解决方案是让应用程序通过 MySQL 客户端接口直接与 MySQL 服务器交互，在其运行过程中的每一个阶段精确地提取出你所需要的信息。

在第 1 章里，列举了几个在 sampdb 示例数据库上需要通过编程与 MySQL 服务器交互来实现的应用目标，下面就是这些目标当中的一些：

- 格式化历史研究会的会员名录输出；
- 提供网上查看和搜索会员名录的服务；
- 发送电子邮件提醒会员交纳会费；
- 使用 Web 浏览器简便地把学生成绩输入成绩册。

需要我们进一步思考的问题是如何把 MySQL 的能力集成到一个 Web 环境里。MySQL 没有为 Web 应用程序提供任何直接支持，但通过把 MySQL 和适当的工具相结合，可以从 Web 服务器代表一个客户端用户发出查询命令并把查询结果返回给那位用户的浏览器。这样一来，通过 Web 去访问数据库的操作就很容易实现了。

MySQL 和 Web 的结合可以从两个相互补充的角度去看。

- **用 Web 服务器提高 MySQL 的可访问性**。从这个角度看问题的人们的主要兴趣是数据库，他们使用 Web 的目的是为了能够更容易地访问数据库里的数据。这应该是 MySQL 系统管理员看问题的角度。此时，数据库是站在台上的主角。比如说，你可以编写一些 Web 页面，它们让你能够看到数据库里的数据表清单，检查每个数据表的结构和内容等。
- **用 MySQL 提高 Web 服务器的数据管理能力**。从这个角度看问题的人们的主要兴趣是 Web 站

点，他们使用 MySQL 的目的是为了让 Web 站点的内容对访问者更有价值。这应该是 Web 开发人员看问题的角度。比如说，如果网站上有一个留言板或论坛，你可以用一个数据库去管理人们发来的消息。此时，MySQL 只是个配角，网站用户甚至有可能根本不会从网站提供的服务里察觉到数据库的身影。

这些观点并不是相互排斥的。比如说，在历史研究会这个例子里，我们将使用 Web 把会员名录发布到网上，让会员能够方便地访问到会员名录的内容。这是使用 Web 来提高数据库可访问性的一个例子。与此同时，把名录内容发布在学会网站上也提高了网站对会员的价值，所以这同时也是使用数据库来提高网站服务水平的例子。

无论如何看待 MySQL 与 Web 的集成，其实现是相似的。以 Web 服务器为桥梁把 Web 站点前端与 MySQL 后端连接在一起。Web 服务器从客户端用户那里收集信息，把它以查询命令的形式发送到 MySQL 服务器，然后再把检索出来的结果返回给客户端的浏览器供用户查看。

当然，完全可以不把你的数据发布到网上，但那么做通常是有好处的，对比通过标准的 MySQL 客户端程序去访问数据的情况就更能体会到这一点。

- 通过 Web 来访问你数据的人可以在他们喜欢的任何一种平台上使用他们喜欢的任何一种浏览器，不必局限于运行着标准 MySQL 客户程序的系统。无论 MySQL 客户的传播范围有多大，Web 浏览器的传播范围都比它大。
- Web 应用程序的操作界面总是可以做到比 MySQL 客户程序的命令行操作界面更简单。
- Web 应用程序的操作界面可以根据具体的应用要求进行定制，而 MySQL 客户端程序都是些通用的工具，其操作界面是固定的。
- 动态 Web 页面扩展了 MySQL 的能力，让我们可以完成许多单独使用标准 MySQL 客户端程序难以或不可能完成的任务。比如说，单独使用 MySQL 客户端程序无法真正实现一个具备“购物车”功能的应用程序。

任何一种程序设计语言都可以用来编写基于 Web 的应用程序，但有些语言要比其他语言更适合这个工作。我们将在 6.3 节讨论这个问题。

6.2　MySQL 应用程序可用的 API

MySQL 服务器有一个底层的“本地”客户/服务器协议，该协议对客户程序应该如何与 MySQL 服务器建立连接和通信做出了定义。客户可以在各种各样的抽象层使用这个协议。

- 为了支持应用程序的开发，MySQL 提供了一个用 C 语言编写的客户端库，它使我们可以在任何一个 C 程序里去访问 MySQL 数据库。这个客户端库实现了一个 API，该接口由一组分别对应着各种本地协议操作的数据结构和函数构成。由这个库提供的 C API 比本地协议更便于使用。
- MySQL 与其他编程语言的接口可以把这个 C 客户端库链接到语言处理器里。利用这个客户端库提供的手段，我们就可以在 C API 的基础上实现 MySQL 与其他程序设计语言的接口。Perl、PHP、Python、Ruby、C++、Tcl 等语言都有这种类型的接口。
- MySQL 与某些编程语言的接口采用了直接实现本地客户/服务器协议的办法来处理通信，没有使用 C 客户端库。Java、PHP 和 Ruby 等语言都有这种类型的接口。

MySQL 与每一种编程语言的接口都定义了它的 MySQL 访问规则。本章的篇幅有限，无法把 MySQL 与其他编程语言之间的 API 全部介绍给大家。我们将把注意力集中在 3 种最流行的 API 身上。

- **C 客户端库 API**。这是最基础的 MySQL 编程接口。MySQL 发行版本里的许多标准化客户程序都是用它来实现的，其中包括 `mysql`、`mysqladmin` 和 `mysqldump`。
- **Perl DBI API**。DBI 是 Database Interface（数据库接口）的缩写。DBI 被实现为一个 Perl 模块，该模块需要与属于 DBD（Darabase Dribver，数据库驱动程序）层的其他模块配合使用，DBD 层的每一个模块提供了相应的数据访问机制。这里将要使用的是专门提供 MySQL 支持的那个 DBD 模块。我们将使用 MySQL 和 DBI 创建几个可以从命令行直接运行的独立脚本和几个用来从 Web 服务器访问 MySQL 的脚本。
- **PHP API**。PHP 是一种服务器端语言，它提供了一种把程序嵌入 Web 页面的简便办法。在被发送给客户之前，这类页面将在服务器主机上由 PHP 进行处理，这就使得脚本可以生成动态内容，例如把 MySQL 查询命令的结果包含到页面里。类似于 DBI，除了 MySQL，PHP 还支持访问另外几种数据库引擎。它既有仅适用于某种特定数据库引擎的接口，也有比较通用的接口。本书将使用后者当中的一种，它叫做 PDO（PHP Data Object）。

为了让你在为某个特定的应用程序挑选编程接口时能够做到心中有数，本章将对这 3 种 API 的基本特点进行对比介绍。随后的 3 章将对它们逐一深入讨论。

当然，根本用不着限制自己只使用一种 API。多了解几种 API 并用知识武装可以在它们当中做出更明智的选择。如果你有一个分成几个部分的大项目，可以选用多种 API，并根据各项具体工作的具体要求选择最适当的语言编写其中一部分，选用另一种语言编写另一部分。如果有足够的时间，不妨尝试以多种方式去实现同一个应用程序，这肯定会让你受益匪浅。这可以让你在使用不同的 API 来编写应用程序的过程中获得最直接的体验。

如果现在还没有使用这几种 API 所必需的软件的话，请阅读附录 A。

如果在学完后面几章内容后还有兴趣了解其他的 MySQL 编程技术，可以自行研读其他书籍。我最熟悉的两本书（因为我就是它们的作者）是 *MySQL and Perl for the Web*（New Rider 出版公司，2001 年）和 *MySQL Cookbook, Second Edition*（O'Reilly 出版公司，2006 年）。第一本书详细讨论了 MySQL 和 DBI 在 Web 环境下的使用技巧，第二本书讨论了如何使用 Perl DBI、PHP PEAR DB 模块、Ruby DBI（类似于 Perl DBI）、Python 的 DB-API 接口、Java JDBC 接口来编写 MySQL 程序。如果对 Java 特别有兴趣，建议阅读 *MySQL and Java Developer's Guide*（Matthews、Cole 和 Gradecki 合著，Wiley 出版公司 2003 年出版），其作者之一 Mark Matthews 是 MySQL“正宗”的 Java 接口 MySQL Connector/J 的开发人员之一。

6.2.1　C API

C API 用在经过编译的 C 程序上下文里。C API 是一个客户端库，它提供了与 MySQL 服务器进行通信的接口，使我们能够与 MySQL 建立连接并进行通信。

MySQL 发行版本里 C 客户程序都基于这个 API。在 MySQL 与其他编程语言的接口当中，有相当一部分是以这个 C 客户端库为基础的。比如说，正是因为链接了这个 MySQL C 客户端库里的代码，与 Perl DBI 模块配套的 MySQL 驱动程序才能与 MySQL 服务器交互。

6.2.2　Perl DBI API

Perl DBI API 用在以 Perl 脚本语言编写的应用程序上下文里。这套 API 要为尽可能多的数据库提供 Perl 编程接口，同时还要让脚本的编写者无需关注与数据库打交道的细节。为实现这一目标，DBI

项目团队设计出了一个由两个层次构成的体系结构（如图 6-1 所示），接口功能由分别位于两个层次的多个 Perl 模块配套实现。

- DBI（database interface，数据库接口）层。负责为客户脚本提供一个通用的接口。这一层与具体的数据库引擎无关，是 DBI 体系结构中的抽象层。
- DBD（database driver，数据库驱动程序）层。以驱动程序（特定于引擎）的形式为各种数据库引擎提供支持。为 MySQL 实现 DBI 支持的 DBD 层模块的名字是 `DBD::mysql`。

DBI 体系架构可以让我们编写出可移植性更好的应用程序。在编写 DBI 脚本时，有一整套标准化的数据库访问调用可供选择。DBI 层将从 DBD 层挑选一个适当的驱动程序去处理请求，那个驱动程序将处理与特定数据库服务器通信的细节问题。从服务器返回的数据将先由 DBD 层传递回 DBI 层，再由后者把数据传递到应用程序。不管数据来自哪一种数据库，数据的格式是统一的。

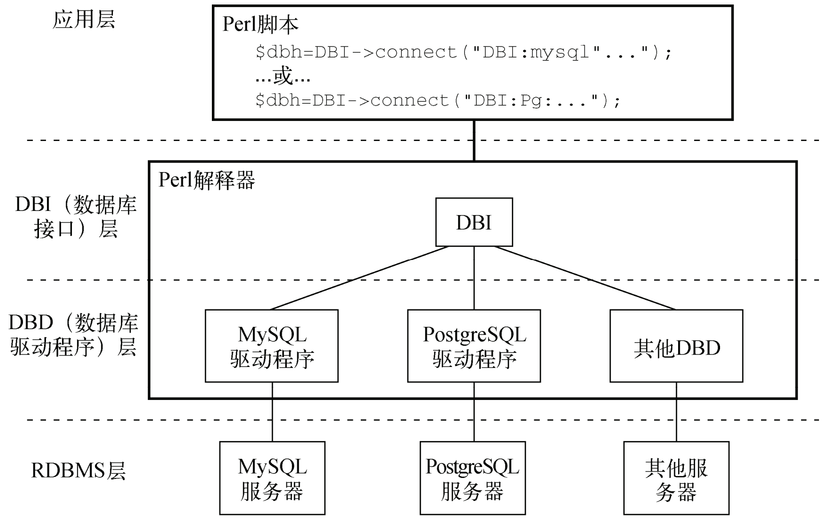


图 6-1 DBI 接口的体系架构

从应用程序开发人员的角度看，DBI 接口掩盖了数据库引擎之间的差异，同时又能适用于多种多样的引擎，和 DBI 驱动程序的种类一样多。DBI 所提供的标准化客户端接口极大地改善了可移植性，使我们能够通过一种方式去访问多种数据库引擎。

在编写 DBI 脚本的时候，只有一个地方需要区别对待不同的数据库引擎，那发生在连接某个特定的数据库服务器的时候，因为你必须告诉 DBI 应该选用哪种 DBD 驱动程序去建立本次连接。比如说，如果是访问一个 MySQL 数据库，就应该使用如下所示的语句来建立连接：

```
$dbh = DBI->connect ("DBI:mysql:...");
```

如果是访问一个 PostgreSQL 或 Oracle 数据库，就应该使用如下所示的语句来建立连接：

```
$dbh = DBI->connect ("DBI:Pg:...");
$dbh = DBI->connect ("DBI:Oracle:...");
```

在建立起连接之后，就再也用不着引用任何驱动程序了。DBI 和驱动程序能够把与特定数据库有关的全部细节处理好。

请注意，这只是理论上的结论。在实际工作中，有两个因素会影响到 DBI 脚本的可移植性。

- 不同的 RDBMS 引擎有不同的 SQL 实现，因而完全有可能出现同一条 SQL 语句在一种引擎上运行良好，但对另一种引擎而言却无法理解的情况。一般来说，如果脚本里的 SQL 语句都足够“正宗”，它就会有足够良好的可移植性。如果脚本里有 SQL 语句使用了只有某种特定的数据库引擎才支持的功能，它的可移植性就要大打折扣。比如说，如果某个脚本里有 MySQL 特有的 `SHOW VARIABLES` 语句，该脚本在其他数据库服务器上就不能正常工作。
- 每一种数据库系统都有它的独到之处，为了让 DBI 脚本的编写者们能够使用这些独特的功能，相应的 DBD 模块特往往会提供一些专用的信息。比如说，对应于 MySQL 数据库引擎的 DBD 模块提供了一些用来访问查询结果里的数据列的属性（如每个数据列里的值的最大长度，它们是不是数值型数据列，等等）的方法，但其他的数据库服务器未必会提供这些信息。这种只有特定的 DBD 驱动程序才具备的功能与可移植性是对立的，如果使用了它们，为 MySQL 编写的脚本就不那么容易用在其他的数据库系统上了。

尽管上述两个因素会影响到 DBI 脚本的可移植性，以抽象层方式提供数据库的 DBI 机制仍是提高可移植性的有效途径。DBI 脚本的编写者可以自己决定是否使用那些会降低可移植性的功能。第 8 章中我并没有刻意回避使用 MySQL DBD 所提供的仅适用于 MySQL 的构造。我那么做是有原因的：要是连那些构造是什么都不知道的话，凭什么去决定要不要使用它们呢？关于这方面的详细讨论见附录 H，那里列出了所有仅适用于 MySQL 的构造。

6.2.3　PHP API

类似于 Perl，PHP 也是一种脚本语言，但 PHP 没有像 Perl 那样被设计成一种通用语言。更准确的说法是，PHP 是一种专门用来编写 Web 应用程序的语言。PHP API 的主要用途是把可执行脚本嵌入 Web 页面。它使 Web 程序员可以很容易地编写出带有动态内容的页面。当某个客户浏览器把一个 PHP 页面请求发送到 Web 服务器时，PHP 将执行该页面里的脚本代码并把它们替换为脚本的输出。结果将被发送到浏览器。这就使得实际显示在浏览器里的页面可以根据当前环境而发生变化。比如说，如果把如下所示的 PHP 脚本嵌入一个 Web 页面，它将显示请求该页面的客户主机的 IP 地址：

```
<?php echo $_SERVER["REMOTE_ADDR"]; ?>
```

举一个有趣的例子，你可以用一个脚本把数据库内容的最新变化情况提供给访问者。如下所示的简单脚本完全可以用在历史研究会的 Web 站点上。这个脚本将发出一条查询命令以确定最新的会员人数并把它报告给这个网站的访问者：

```
<html>
<head>
<title>U.S. Historical League</title>
</head>
<body bgcolor="white">
<p>Welcome to the U.S. Historical League Web Site.</p>
<?php
# USHL home page

try
{
  $dbh = new PDO("mysql:host=localhost;dbname=sampdb", "sampadm", "secret");
  $dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

```
    $sth = $dbh->query ("SELECT COUNT(*) FROM member");
    $count = $sth->fetchColumn (0);
    print ("<p>The League currently has $count members.</p>");
    $dbh = NULL;  # close connection
  }
  catch (PDOException $e) { } # empty handler (catch but ignore errors)
  ?>
  </body>
  </html>
```

PHP 脚本看上去很像 HTML 页面，只是多了一些嵌在<?php 和?>标签之间的可执行代码。每个页面可以包含任意多个代码段。这就提供了一种极其灵活的脚本开发手段。比如说，可以像编写一个普通 HTML 页面那样编写一个 PHP 脚本以搭建该页面的基本框架，然后再添加代码去生成该页面里的动态内容。

PHP 有多种数据库接口。早期的 PHP 数据库接口由一组底层的函数库构成，每个函数库对应着一种数据库引擎，但不像 DBI 接口那样下功夫对不同引擎的接口进行标准化。所以那些接口没有统一的调用形式，看起来更像是用 C 语言为某种数据库引擎实现的底层 API 函数库。比如说，用来从 PHP 脚本访问 MySQL 数据库的 PHP 函数的名字和 MySQL C 客户端库里的函数的名字非常相似。

PHP 也有更像 DBI 的数据库接口，它们是用 PDO（PHP 数据对象，PHP Data Object）扩展模块实现的。这个扩展模块通过使用类似于 DBI 的双层体系架构而提供了一种更抽象的数据库引擎接口。第 9 章在访问数据库时使用的是 PDO 扩展模块。

6.3　如何挑选 API

本节提供了一些通用的指南以帮助大家为不同类型的应用程序挑选 API。我们将对比 C、DBI 和 PHP API 的能力以便让大家对它们的相对优点和缺点有所了解，还将给出一些关于何时应该选择何种 API 的建议。

虽然我在这几种语言当中有自己的偏好，但我没有偏见。你肯定也有自己的偏好，就像本书的技术校对编辑那样：他们当中有一位认为我应该大力强调用 C 语言编写 MySQL 程序的重要性，另一位却认为我应该压缩 C 语言部分的篇幅，并建议大家尽量避免使用这种语言。从这些迥然不同的观点可以得到这样一个启示：应该认真思考本节所讨论的因素并得出自己的结论。

在评估哪一种 API 最适合某个特定的任务时，有许多因素需要考虑。

- **预期的执行环境**。编写的应用程序将在什么样的上下文里使用。
- **性能**。如果使用某种 API 语言来编写应用程序，它的执行效率将有多高。
- **开发工作的难易程度**。使用某种 API 及相关语言来编写应用程序是否方便快捷。
- **可移植性**。这个应用程序会不会用于 MySQL 以外的其他数据库系统。

接下来的讨论将依次对这几个因素做进一步分析。请注意，这几个因素是相互影响的。比如说，谁都希望自己编写的应用程序有高性能，但在某些场合，选用某种编程语言的首要条件可能是它能让你迅速完成应用程序的开发工作而不是单纯追求性能。

6.3.1　执行环境

在开始编写一个应用程序的时候，通常对它的使用环境已经有了大致的了解。比如说，接到的任务或许是编写一个需要从 shell 启动运行的报表生成程序，或许是一个将在每个月的月底作为 cron 任

务运行的应付账款汇总程序。而从 shell 启动运行或是作为 cron 任务运行的命令，通常需要编写成一个无需其执行环境提供很多信息的独立程序。再比如说，你接到的任务也许是编写一个供 Web 服务器调用的应用程序，这类程序往往需要从其执行环境了解很多非常具体的信息：客户正在使用的浏览器是哪一种？访问者在电子订阅申请表单里输入了什么样的参数？客户为访问个人信息而输入的口令是否正确？

每种 API 都有适合用它来编写的应用程序，而且情况还会随着环境的变化而变化。

- C 语言是一种通用语言，所以从理论上讲，它可以用来编写任何应用程序。但在实际工作中，C 语言更适合用来编写独立程序，不太适用于 Web 编程。这或许是因为在 C 程序里进行文本处理或内存管理不像 Perl 或 PHP 那样容易，而 Web 应用程序往往需要频繁使用这些功能。
- 类似于 C 语言，Perl 也适合用来编写独立程序。但碰巧 Perl 在 Web 站点环境里也非常有用，例如需要借助于 CGI.pm 之类的扩展模块。这就使得 Perl 很适合用来编写需要把 MySQL 和 Web 结合在一起的应用程序。这样的程序可以通过 CGI.pm 模块与 Web 交互，通过 DBI 接口与 MySQL 交互。
- PHP 是人们为了编写 Web 应用程序而设计的一种程序设计语言，因而最适合在 Web 环境里使用。不仅如此，数据库访问也是 PHP 的强项之一。因此，当准备编写一个需要执行 MySQL 数据库查询任务的 Web 应用程序时，Perl 会是一个很不错的选择。我们完全可以把 PHP 当做一个独立的语言解释器来使用（比如用它从 shell 执行一个脚本），但这种用法比较少见。

综合以上分析，C 和 Perl 是编写独立应用程序时的最佳候选语言。对于 Web 应用程序，还是用 Perl 或 PHP 最适合。如果需要同时编写这两种类型的应用程序，而你对这些语言又都不太熟悉，Perl 应该是最佳选择，因为它需要你学习的东西最少。

6.3.2　性能

如果其他方面都一样，我们当然更喜欢运行速度最快的应用程序。但在实际工作中，性能的重要性往往与程序的使用频率有关。对于一个每个月只在后半夜作为 cron 任务运行一次的应用程序来说，它的性能对全局几乎没有什么影响。但从另一个方面看，如果某个程序会在一个访问量极大的 Web 站点上每秒运行许多次的话，对其性能的一丁点儿改善都将导致显著的差异。在后一种情况里，性能对网站的可用性和受欢迎程度有着巨大的影响。不管它提供的内容是什么，一个反应迟钝的网站肯定不会受到人们的欢迎。如果依赖这样的网站作为收入来源，性能的低劣将直接转化为利润的减少。既然网站不能同时支持尽可能多的连接，就不要抱怨那些等得不耐烦的访问者会另寻他处了。

性能评估是一个复杂的问题。如果想知道用某种 API 编写的应用程序会有怎样的性能，最好的办法是使用该 API 把它编写出来并测试它。最好的性能对比测试方案是用不同的 API 实现出多个不同的版本、再让它们一对一决出高下。通常先把程序编写出来，等它能用以后再去考虑如何进行优化才能让它运行得更快、占用内存更少以及是否还有其他方面可以改进。无论如何，至少有两个值得关注的基本因素会对性能产生相对持久的影响。

- 编译出来的程序要比以解释方式执行的脚本执行得更快。
- 对于用在 Web 上下文里的解释型语言，让解释器作为 Web 服务器的一个模块来运行要比作为另外一个进程来运行的性能更好。

1. 编译型语言与解释型语言

一般而言，如果功能完全一样，编译而来的应用程序要比使用脚本语言编写的版本更有效率，使

用的内存更少、执行得更快。这是因为用来执行脚本的语言解释器可以导致相当大的开销。C 是一种编译型语言，Perl 和 PHP 都是解释型语言，所以 C 程序一般比 Perl 或 PHP 脚本运行得更快。因此，C 语言应该是编写使用频率很高的程序时的最佳选择。

不过，还有其他一些因素会缩小编译型程序与解释型程序之间的性能差距。首先，用 C 语言编写出来的程序通常都运行得很快，但效率低劣的 C 程序也并不罕见。用编译型语言编写出程序并不能保证一定会有更好的性能，仍需要根据具体情况来决定如何去做。其次，如果某个脚本型应用程序大部分时间是在执行那些经过编译并被链接到解释器引擎里的 MySQL 客户端库里的代码，编译型程序和解释型程序的性能差距就会小得多。

2. 语言解释器：独立版本与模块版本

对于基于 Web 的应用程序而言，其脚本语言解释器通常有两种使用方式，至少对 Apache 来说是这样，而这本书里用来编写和运行 Web 应用程序的 Web 服务器正是 Apache。

- 可以安排 Apache 把脚本解释器作为另外一个进程调用。在这种操作模式下，当 Apache 需要运行一个 Perl 或 PHP 脚本时，它将启动相应的解释器并让它去执行那个脚本。此时，Apche 是把解释器当做 CGI 程序来使用。换句话说，它将使用通用网关接口（Common Gateway Interface，CGI）协议与它们通信。
- 解释器还可以作为一个被直接链接到 Apache 的二进制可执行代码里的模块来使用，该模块将作为 Apache 进程的一部分而运行。用术语来说，Perl 和 PHP 解释器将以 Apache 服务器的 `mod_perl` 和 `mod_php` 模块的形式运行。

Perl 和 PHP 的支持者都宣称自己喜欢的解释器有速度优势，但一致同意这两种解释器的运行方式要比这两种语言本身对性能的影响更大。与作为独立的 CGI 程序运行时的情况相比，这两种解释器在以模块方式运行时的速度快了很多。如果把它们运行为独立的应用程序，每执行一个脚本，就必须启动它们一次，因此而产生的进程创建开销是相当巨大的。如果把它们当做一个已经在运行的 Apache 进程的模块来使用，Web 页面在任何时候都可以立即访问解释器的功能。开销的减少使性能获得巨大改善，因而能够同时处理更多的访问请求和更快地完成对它们的处理。

与模块版本相比，独立型解释器在启动时的开销往往会导致性能降低一个数量级。不仅如此，通过 Web 页面提供的服务大都是一些计算量很小的快速事务，不需要进行大量的计算和处理，如果把这个因素也考虑进来，解释器的启动开销就更是一减再减了。如果花费大量的时间去启动解释器，但只花费很少的时间执行脚本，绝大多数资源就白白浪费了。这就好比用了几乎一整天的时间去做准备工作，结果却是下午 4 点到田里，不到 5 点就回家了。

你也许会对解释器的模块版本为什么能改善性能感到不解。不管怎么说，你总得启动 Apache 本身，对吧？性能方面的改善来自这样一个事实：一个给定的 Apache 进程可以同时处理多个请求。在 Apache 启动的时候，它会立刻“繁衍”出一批子进程用以处理访问请求。当一个需要执行一段脚本代码的请求到达时，早有一个 Apache 进程在等着处理它了。还有，每个 Apache 实例都能同时处理多个请求，所以其进程启动开销将为每批访问请求发生一次，不是每个请求发生一次。（Apache 2 在必要时可以使用多个线程而不是另外再去创建一个 Apache 进程，所以它在这方面的开销就更小了。）

Perl 和 PHP 的另一个主要区别是 Perl 需要占用更多的内存。链接了 `mod_perl` 模块的 Apache 进程要比链接了 `mod_php` 模块的更大一些。PHP 的设计理念之一是它必须在另一个进程的内部生存并且在其中的生命期里可能会“死而复生”许多次。Perl 的设计理念则是从命令行作为一个独立的程序来运行，当初并没有考虑到把它嵌入到一个 Web 服务器进程里作为一种语言执行机制。这或许就是 Perl

需要占用更多内存的原因之一：在作为模块运行的时候，其运行环境并不是专门为它准备的。导致 Perl 占用更多内存的其他原因还包括脚本缓存功能，以及脚本需要额外使用其他的 Perl 模块。在发生这两种情况的时候，会有更多的代码进入内存并待在那里直到 Apache 进程结束。（为了把这个问题降到最低，有一些办法可以让你指定只有某几个特定的 Apache 进程才允许启动 `mod_perl` 模块。这样一来，只有那些允许执行 Perl 脚本的进程才会发生额外的内存开销。Apache 官方网站上的 `mod_perl` 专区对各种可供选择的策略进行了深入的讨论；详见 http://perl.apache.org/docs/。）

语言解释器的独立版本至少有一个超越其模块版本的优点：可以安排它在一个不同的用户 ID 下运行脚本。模块版本只能使用与 Web 服务器相同的用户 ID 运行脚本，出于安全方面的考虑，那往往是一个权限最小的账户。这会让那些需要特殊权限的脚本（例如，读或写受保护的文件）难以工作。如果愿意，可以让模块版本和独立版本结合起来使用：默认使用模块版本，在遇到脚本需要某个特定用户的特殊权限才能运行时切换到独立版本。

综上所述，无论选择 Perl 还是 PHP，都应该尽量把它用作 Apache 模块，而不是调用一个解释器进程。只有当解释器的模块版本无能为力的时候（例如，执行某个需要特殊权限才能正常工作的脚本）才切换到其独立版本。在遇到这类情况的时候，可以通过使用 Apache 的 suEXEC 机制在某个给定的用户 ID 下启动解释器来处理脚本。

6.3.3　开发时间

我们刚刚讨论了一些会影响应用程序性能的因素，但单纯追求执行效率不应该是唯一目标。你自己的时间以及编程工作的难易程度也很重要。为 MySQL 应用程序挑选 API 的时候，需要多长时间才能完成开发工作也是一个必须考虑的因素。如果只用花费开发 C 程序所需时间的一半就可以开发出一个功能与之完全相同的 Perl 或 PHP 脚本，就算最终的应用程序运行得没那么快，仍坚持选用 C API 的人也不会很多。在某些场合，不太在意应用程序的执行时间，多关注一下编写应用程序所花费的时间也许更有道理，尤其是在那个应用程序的使用频率并不是很高的时候。你个人的一个小时要比计算机的一个小时宝贵得多！

一般来说，脚本语言可以让你更快地完成程序的编写工作，尤其是在为最终的应用程序搭建原型的时候。这至少有两个原因。首先，脚本语言提供的语法结构通常都要比编译语言提供的更高级。这使你能够从一个更高的抽象层来思考问题，把注意力集中在“应该做什么”的大方向而不是“如何去做”的小细节上。比如说，在与涉及“键/值”关系（例如“学生 ID/学生姓名”）的数据交互时，PHP 语言中的关联数组和 Perl 语言中的散列可以帮助程序员节约大把的时间。C 语言没有提供任何类似的构造。如果坚持使用 C 语言来实现它的话，将不得不编写代码去处理诸如内存管理、字符串操作之类的底层细节，代码写完后还必须调试它们。这都得花费时间。

其次，使用脚本语言的程序开发周期比使用编译语言的少。如果使用 C 语言，程序开发周期将由“编辑–编译–测试”3 个环节组成。每修改一次程序，就必须重新编译它，然后再测试。如果使用 Perl 或 PHP 语言，程序开发周期简化成“编辑–测试”两个环节，脚本修改后无需编译就可以立刻上机测试。从另一方面看，编写 C 代码有更多的约束，C 编译器也会对程序做更严格的类型检查。编译器对代码的检查越严格，就越容易发现可能存在的编程漏洞，较为宽松的语言（如 Perl 和 PHP）容易在代码里留下隐患。如果在 C 程序里拼错了一个变量名，编译器会向你报警。PHP 和 Perl 却要等你让它们这么做时才会这么做。当应用程序变得越来越大、越来越难以维护的时候，这些更加严格的约束将越能体现其价值。

一般来说，在编译语言和解释语言之间做出选择的决定性因素是开发时间和性能。想用编译语言开发程序吗？它执行得更快，但需要花费更多的时间来编写。想把程序编写为一个脚本吗？可以用最短的时间让它运行起来，但在执行速度方面要付出一些代价。

还可以把这两种办法结合起来。先编写一个脚本作为应用程序的“草稿”，用这个快速搭建起来的原型去检验你的逻辑思路和你选用的算法是不是最好。如果事实证明这个程序很有用，并且它的使用频率也高到了需要改善其性能的程度，那就把它改写为一个编译型应用程序好了。这是一个两全其美的办法：既可以在初始开发阶段快速建立原型，又能让最终产品获得最好的性能。

从严格的意义上讲，Perl DBI 和 PHP API 都不能提供任何在 C 客户端库里找不到的功能。可是，用 C 语言往应用程序里嵌入 MySQL 数据库访问功能，与用 Perl 或 PHP 语言这么做的开发环境是非常不同的。在面临选择的时候，应该向自己提出并回答两个问题：这个程序在和 MySQL 服务器交互的时候将需要完成哪些任务？每一种 API 可以在多大程度上帮助完成这些任务？下面是一些例子。

- **内存管理**。在 C 语言里，只要涉及动态分配的数据结构，就需要和 `malloc()` 和 `free()` 函数打交道。Perl 和 PHP 则能够替你处理好这些事情。比如说，Perl 和 PHP 都允许数组的长度自动增加，都允许使用长度可变的字符串而无需考虑内存管理问题。
- **文本处理**。Perl 在这一领域的能力最强，PHP 以微弱差距位居第二。C 语言和它们相比只能算是非常初级的水平，远远地落在第三名的位置上。

当然，在 C 语言编程环境里，可以通过自行创建函数库、把诸如内存管理和文本处理之类的任务封装成函数让工作变得简单些。但这又增加了调试的负担，而且还必须保证选用的算法都足够好。在这些方面，Perl 和 PHP 里的这类算法至少有这样一点优势：因为已经有足够多的眼睛检查过那些算法，所以它们应该已经接受过足够好的调试并有着足够好的性能。既然早已有人把时间花费在了这些方面，为什么不采用“拿来主义”节约你自己的时间呢？（从另一方面讲，万一在你使用的解释器里存在着一个 bug，在问题解决之前，你只能忍气吞声或是绕道而行。换成用 C 语言编写的程序，你可以对程序的行为做出更细致的控制。）

程序设计语言在安全性方面也是各有千秋。C API 提供的数据库服务器接口是最底层的，也是安全策略最少的。从这个意义上讲，它提供的安全保障也是最少的。如果调用 C API 函数的顺序不对头，运气好的话只会看到一条“同步错误”出错消息，运气差的话就只能眼看着程序崩溃了。Perl 和 PHP 都可以把你保护得相当好。即使做事情的顺序不对头，脚本运行失败，解释器也不会发生崩溃。动态分配内存以及相关指针的使用也很容易在 C 程序里留下崩溃 bug。Perl 和 PHP 都能替你把内存管理好，所以脚本不太可能因为内存管理方面的 bug 意外“死亡”。

除了上面讨论的几个因素以外，开发时间的长短还与程序设计语言的外部支持的多少有关。C 语言的外部支持以各种函数封装库的形式出现，它们把 MySQL C API 库里的函数封装成一系列更容易使用的例程。C 和 C++都有函数封装库形式的外部支持。Perl 的外部支持无疑是最多的，它们以 Perl 模块的形式出现（这些模块在概念上类似于 Apache 模块）。甚至有人专门组建了一个档案网站（CPAN，Comprehensive Perl Archive Network，Perl 综合档案网）来方便人们寻找和获得这些模块。利用 Perl 模块，程序员只需填写几个参数就可以使用各种各样的函数——甚至连一行完整的代码都不用编写。想编写脚本从数据库生成一份报告并把它作为附件发送给什么人吗？没问题，访问 cpan.perl.org 网站，挑选一个 MIME 模块下载回来，就能立刻获得这种附件自动生成能力。PHP 也有它自己的档案网站，比较著名的有 PEAR（PHP Extension and Application Repository，PHP 扩展模块和应用程序仓库）和 PECL（PHP Extension Community Library，PHP 扩展模块公共档案馆），它们的网址分别是 pear.php.net

和 pecl.php.net。

6.3.4　可移植性

可移植性可以归结为这样一个问题：把一个为了访问 MySQL 数据库而编写的程序改成可以用于另外一种数据库引擎，修改工作的难易程度有多大？你也许从未认真思考过这个问题。除非能预见未来，否则说“我绝不会把这个程序用在 MySQL 以外的任何数据库上”这样的话就是在冒险。假设你刚换了个新工作并打算继续使用你的老程序，可新东家使用的是另外一种数据库系统，你该怎么办？如果对可移植性有要求，就必须了解不同的 API 在这方面的差异。

- DBI API 的可移植性最高，因为与数据库无关正是 DBI 明确的设计目标之一。
- 使用前面提到的 PDO 数据库接口扩展模块编写出来的 PHP 脚本的可移植性接近于 DBI。如果只使用了底层的数据库接口函数库，如此编写出来的 PHP 脚本的可移植性要差一些，因为那些接口函数库不像 DBI 那样对各种数据库引擎的访问接口进行过统一。那些接口库里的 PHP 函数的名字大都沿用了相应的底层 C API 库里的函数名。因此，如果打算改用另外一种数据库引擎，至少需要把 PHP 脚本里那些与数据库有关的函数的名字改过来。也许还需要对应用程序的逻辑稍微做一些修改，因为不同的数据库在接口的工作流程方面往往会有所不同。
- 使用 C API 编写出来的应用程序在数据库之间的可移植性最差。这是因为 C API 是专门为了访问 MySQL 数据库而设计的，根本没有考虑到可移植性的问题。

如果需要在同一个应用程序里访问多个不同种类的数据库系统，从“程序代码与数据库无关”这个意义上讲的可移植性将尤其重要。这也许只是些简单的任务，比如把数据从一个 RDBMS 迁移到另外一个；也许会是些相当复杂的任务，比如把来自多个不同种类的数据库系统的信息整合在一起生成一份报告。

第 7 章

用 C 语言编写 MySQL 程序

MySQL 提供了一个用 C 语言写成的客户端库，你可以用来编写能够访问 MySQL 数据库的客户程序。这个库定义了一套应用程序编程接口，其中包括以下几大类例程。

- 连接管理例程，用来建立和断开与 MySQL 服务器的会话。
- 构造 SQL 语句并发送给服务器，然后处理结果的例程。
- 状态检查和出错报告类函数，用来在 API 函数调用失败时了解出错的原因。
- 选项处理类例程，帮你处理选项文件或命令行上的选项。

本章演示如何使用 C 客户端库编写程序，我们将沿袭与 MySQL 发行版本所收录的那些“标准”客户程序大致相同的体例和风格。我将假设你对如何使用 C 语言进行编程有一定程度的了解，但并不是一位专家。

本章的第一部分将开发一系列短小的程序，最终成果是一个用来充当 MySQL 客户端编程框架的简单程序，其功能仅限于与服务器建立和断开连接。这么做的理由是：虽然 MySQL 客户程序各有各的用途，但它们的一个共同之处是都必须与服务器建立连接。

最终得到的主干程序有着很好的通用性，可以把它用做开发其他客户程序的基础。在完成了它的开发工作之后，我们将先学习如何执行各种 SQL 语句。我们将首先探讨如何处理硬编码的 SQL 语句，然后再去开发一段能够用来处理各种 SQL 语句的代码。在完成这部分的学习之后，我们将给主干程序增加一些语句处理代码，如此开发出来的程序与 MySQL 自带的 `mysql` 客户程序非常相似，你可以用它来交互地发出语句。

接下来，本章将演示另外几种可以借助于 MySQL C 客户端库实现的功能。

- 编写使用安全套接字层（Secure Socket Layer，SSL）协议通过安全连接与 MySQL 服务器通信的客户程序。
- 编写使用嵌入式服务器开发库 libmysqld 的应用程序。
- 向服务器一次发送多条语句，并对返回的结果集进行处理。
- 使用服务器端预处理语句。

本章只讨论来自客户端库的函数和数据类型（示例程序会用到），它们的完整清单请见附录 G。附录 G 对客户端库的其他方面也做了一些介绍，你可以在开发工作中随时参考。

本章中的示例程序都可以从网上获得，你可以直接运行它们而不必亲自键入。它们都收录在 sampdb 发行版本的 capi 子目录里。具体的下载办法请参见附录 A。

到哪里去找示例程序

在 MySQL 邮件列表上经常会看到这样的问题："在哪里能找到用 C 语言写的客户程序？"如果让我来回答这个问题，那当然是："在这本书里！"不过，似乎有很多人都没注意到，在 MySQL 自带的客户程序里就有好几个（例如 mysql、mysqladmin 和 mysqldump）是用 C 语言写出来的。因为这种发行版本是现成的，所以它能提供相当多的示例客户代码。如果你还没有这样做过，那就赶快下载一份源代码发行版本并看看它的 client 和 tests 子目录里的程序吧。

7.1 编译和链接客户程序

利用 MySQL C 客户端库编写出来的程序要经过编译和链接才能执行，本节将对这些步骤进行介绍。不同的系统用来构建客户程序的命令往往会有一些差异，本节里的命令可能需要稍做修改才能用在你的系统上。不过，基本原则是通用的，你编写的大部分客户程序都可以使用。

既然是用 C 语言编写 MySQL 客户程序，那你肯定需要 C 语言编译器。本节的示例将使用 gcc 编译器，它是 Unix 系统上最常见的编译器。此外，你还需要 MySQL 头文件和客户端库。

头文件和客户端库是开发 MySQL 客户程序的基础。如果你系统上的 MySQL 软件是从它的一个源代码或二进制发行版本安装的，那么客户程序开发支持组件就应该已经作为安装过程的一部分而安装好了。如果你是用 RPM 文件来安装 MySQL 软件的，就必须安装开发者 RPM 文件才能把这些支持组件安装到你的机器里。MySQL 头文件和客户程序开发库的获取办法请参见附录 A。

在编译和链接客户程序时，经常需要把 MySQL 头文件和客户端库的存放位置明确地告诉给编译器，因为它们的安装位置通常不在编译器或链接器的默认搜索路径上。在下面的例子里，我们将假设 MySQL 头文件和客户端库分别安装在/usr/local/include/mysql 和/usr/local/lib/mysql 子目录里。当然，你可以根据自己的系统修改路径。

在把源文件编译为目标文件时，要用-I 选项把 MySQL 头文件的存放位置告诉给编译器。比如说，要把源文件 myclient.c 编译为目标文件 myclient.o，应该使用以下命令：

```
% gcc -c -I/usr/local/include/mysql myclient.c
```

在把目标文件链接到可执行代码时，要用-L/usr/local/lib/mysql 和-lmysqlclient 参数把客户端库的存放位置和名称告诉给链接器，如下所示：

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient
```

如果客户程序由多个文件组成，就要在链接命令行上把所有的目标文件列举出来。

链接步骤经常会因"某函数没有定义"之类的错误而失败，此时，你需要增加一个-l 选项把"某函数"所在的库的名字告诉给链接器。如果这类出错信息与 compress()或 uncompress()函数有关，就要用-lz 或-lgz 参数来告诉链接器 zlib 压缩工具库的位置：

```
% gcc -o myclient myclient.o -L/usr/local/lib/mysql -lmysqlclient -lz
```

如果这类出错信息与 floor()函数有关，就要用-lm 参数来链接数学函数库。也可能还需要增加其他函数库，例如，在 Solaris 系统上可能还需要-lsocket 和-lnsl 参数。

你可以利用 mysql_config 工具程序来确定 MySQL 程序的编译和链接参数。比如说，这个工具程序可能会给出如下所示的输出：

```
% mysql_config --include
-I'/usr/local/mysql/include/mysql'
% mysql_config --libs
-L'/usr/local/mysql/lib/mysql' -lmysqlclient -lz -lcrypt -lnsl -lm
```

如果想把 mysql_config 直接用在编译或链接命令里，就需要把它放到一对反引号里：

```
% gcc -c `mysql_config --include` myclient.c
% gcc -o myclient myclient.o `mysql_config --libs`
```

系统将先执行 mysql_config 并用执行结果替换反引号中的命令，自动地为 gcc 提供必要的编译或链接参数。

如果你还没使用 make 命令建立过程序，赶快补一下课吧，你将不必手动输入大量的命令。假设你正在开发一个名为 myclient 的客户程序，它由两个源文件 main.c 和 aux.c 以及一个头文件 myclient.h 组成。你可以用一个简单的 Makefile 文件去建立这个程序，如下所示。注意：Makefile 文件中的行缩进必须用制表符来实现，如果你使用的是空格，Makefile 文件就不能工作了。

```
CC = gcc
INCLUDES = -I/usr/local/include/mysql
LIBS = -L/usr/local/lib/mysql -lmysqlclient

all: myclient

main.o: main.c myclient.h
	$(CC) -c $(INCLUDES) main.c
lib.o: lib.c myclient.h
	$(CC) -c $(INCLUDES) lib.c

myclient: main.o lib.o
	$(CC) -o myclient main.o lib.o $(LIBS)

clean:
	rm -f myclient main.o lib.o
```

有了 Makefile 文件，修改源文件后你只需运行 make 就可以重建程序；make 将显示并执行必要的命令：

```
% make
gcc -c -I/usr/local/mysql/include/mysql myclient.c
gcc -o myclient myclient.o -L/usr/local/mysql/lib/mysql -lmysqlclient
```

与键入长长的 gcc 命令相比，这种办法既简单又不容易出错。Makefile 文件还使编译工作更容易控制和管理。例如，如果需要在链接步骤增加几个函数库，如数学函数库和压缩工具函数库，你只需在 Makefile 文件中的 LIBS 行追加上-lm 和-lz：

```
LIBS = -L/usr/local/lib/mysql -lmysqlclient -lm -lz
```

如果还需要用到其他函数库，那就把它们也追加到 LIBS 行上。此后，当你运行 make 命令时，它将自动使用 LIBS 行的新值。

除直接编辑 Makefile 文件外，还有一种办法可以改变 make 变量：在命令行上设定。例如，假如你的 C 编译器名为 cc 而不是 gcc，你可以这样做：

```
% make CC=cc
```

如果你有 mysql_config 工具程序，可以把它用在 Makefile 文件里，这样，你就用不着把头文件和函数库的路径名编码在 Makefile 文件里了。你可以使用如下所示的 INCLUDES 和 LIBS 行：

```
INCLUDES = ${shell mysql_config --include}
LIBS = ${shell mysql_config --libs}
```

当你发出 make 命令时，它会执行每个 mysql_config 命令并且用执行结果作为相应的变量值。${shell}语法是 GNU make 支持的；如果你的 make 版本与 GNU make 无关，就可能需要使用另一种稍微不同的语法。

如果你正使用着一个 IDE（Integrated Development Environment，集成开发环境），你可能根本就不会用到 Makefile 文件，这要取决于你具体使用的是哪一种 IDE。

7.2　连接到服务器

我们的第一个 MySQL 客户程序简单到了极点——它连接一个服务器、断开连接、退出。这一系列动作本身并没有多大的用处，但你必须把它们弄明白，因为如果不能连接到服务器，你就无法对任何一个 MySQL 数据库进行访问。连接服务器是一个非常常见的操作，我们在这里开发的建立连接的代码肯定要出现在你编写的每一个客户程序里。此外，这个任务也使我们能以一个比较简单的问题作为出发点。我们将陆续对这个客户程序作一些改进，让它能够完成一些更有用的事情。

我们的第一个客户程序 connect1 的代码全部放在一个名为 connect.c 的源文件里：

```
/*
 * connect1.c - connect to and disconnect from MySQL server
 */

#include <my_global.h>
#include <my_sys.h>
#include <mysql.h>

static char *opt_host_name = NULL;     /* server host (default=localhost) */
static char *opt_user_name = NULL;     /* username (default=login name) */
static char *opt_password = NULL;      /* password (default=none) */
static unsigned int opt_port_num = 0; /* port number (use built-in value) */
static char *opt_socket_name = NULL;  /* socket name (use built-in value) */
static char *opt_db_name = NULL;       /* database name (default=none) */
static unsigned int opt_flags = 0;     /* connection flags (none) */

static MYSQL *conn;                    /* pointer to connection handler */

int
main (int argc, char *argv[])
{
  MY_INIT (argv[0]);
  /* initialize client library */
  if (mysql_library_init (0, NULL, NULL))
  {
    fprintf (stderr, "mysql_library_init() failed\n");
    exit (1);
  }
  /* initialize connection handler */
  conn = mysql_init (NULL);
```

```
  if (conn == NULL)
  {
    fprintf (stderr, "mysql_init() failed (probably out of memory)\n");
    exit (1);
  }
  /* connect to server */
  if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
      opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
  {
    fprintf (stderr, "mysql_real_connect() failed\n");
    mysql_close (conn);
    exit (1);
  }
  /* disconnect from server, terminate client library */
  mysql_close (conn);
  mysql_library_end ();
  exit (0);
}
```

这个源文件首先把头文件 `my_global.h` 和 `mysql.h` 包括了进来。根据 MySQL 客户程序的用途，它可能还需要再把其他头文件包括进来，但下面 3 个是最基本的。

- `my_global.h` 文件。它负责把其他几个常用的头文件（如 `stdio.h`）包括到源文件里。如果你正在 Windows 系统上编译这个程序，它还会把 Windows 兼容信息也包括进来。（你本人可能不会在 Windows 下编译程序，但如果你打算对外发布代码，把 `my_global.h` 文件也包括进来将对那些需要在 Windows 下编译的人们有帮助。）
- `my_sys.h` 文件。包含着为了提高代码的可移植性而定义的各种各样的宏，以及 MySQL C 客户端库所用到的各种结构和函数的定义。
- `mysql.h` 文件。它定义了基本的 MySQL 常数和数据结构。

文件的顺序非常重要：`my_global.h` 应该在与 MySQL 有关的任何其他头文件之前被包括到源文件里。

接着，程序声明了一组变量，依次对应着将被用来连接 MySQL 服务器的各种参数。在这个客户程序里，这些参数都硬编码在代码里并且都有默认值。稍后，我们将使用一种更灵活的办法来处理这些参数，即允许来自选项文件或命令行的值覆盖这些默认值。（这也正是变量名全都以 `opt_` 开头的原因，这个前缀的意思是那些变量最终将通过命令选项来设定。）程序还定义了一个指向一个 `MYSQL` 结构的指针，这个 `MYSQL` 结构将充当连接处理程序。

程序的 `main()` 函数负责建立和断开与服务器的连接。建立连接需要两个步骤：

(1) 调用 `mysql_init()` 函数获得一个连接处理程序；

(2) 调用 `mysql_real_connect()` 函数建立与服务器的连接。

当你把 `NULL` 传递给 `mysql_init()` 函数时，它将自动分配一个 `MYSQL` 结构，初始化之，然后返回一个指向它的指针。`MYSQL` 数据类型是一个用来保存与连接有关的信息的结构。这类变量称为连接处理程序。

另一种做法是传递一个指向某个现有 `MYSQL` 结构的指针。如果是这样，`mysql_init()` 函数将不用创建结构本身对那个现有的 `MYSQL` 结构进行初始化并返回一个指向它的指针即可。

`mysql_real_connect()` 函数的参数有很多，它们的含义如下所示。

- 指向连接处理程序的指针，它必须是 `mysql_init()` 函数的返回值。

- 服务器主机，对这个值的解释与具体的操作平台有关。在 Unix 系统上，如果你给出的是一个字符串形式的主机名或者数字形式的 IP 地址，客户程序将使用 TCP/IP 连接去连接该主机。如果你给出的是 NULL 或"localhost"，客户程序将使用 Unix 套接字文件去连接运行在本地主机上的 MySQL 服务器。

除 localhost 外，Windows 系统对这个参数也会做出类似的解释，但会用 TCP/IP 连接来代替 Unix 套接字文件连接。此外，在 Windows 系统上，如果你给出的主机名参数是"."或 NULL，并且服务器支持命名管道连接，客户程序将首先尝试使用命名管道去连接本地服务器。

- 连接操作所使用的 MySQL 账户的用户名和口令。如果用户名是 NULL，客户端库将把你的登录名发送给服务器（对于 Windows 系统则是 ODBC）。如果口令是 NULL，则不发送任何口令。
- 将在连接建立后被选取为默认数据库的数据库的名字。如果这个值是 NULL，则不选取数据库。
- 端口号。端口号是供 TCP/IP 连接使用的。值 0 表示让客户端库使用默认端口号。
- 套接字文件名。套接字文件名是供 Unix 套接字文件连接（对于 Unix 系统）或命名管道连接（对于 Windows 系统）使用的。如果这个参数的值是 NULL，客户端库将使用默认的套接字（或命名管道）名。
- 标志值。connect1 程序传递给这个参数的值是 0，因为它没有用到任何特殊的连接选项。

你可以在附录 G（在线资料）里找到对 mysql_real_connect()函数的详细介绍。附录 G 对主机名参数对端口号和套接字文件名参数的影响、允许用在连接标志参数里的各种选项作了进一步的说明。附录 G 还介绍了 mysql_options()函数的用法。在调用 mysql_real_connect()函数之前，你可以用 mysql_options()函数对其他一些与连接有关的选项进行设置。

调用 mysql_close()函数并把指向某个连接处理程序的指针传递给它就可以断开这个连接。如果这个连接处理程序是你当初通过把 NULL 传递给 mysql_init()而自动分配的，mysql_close()调用将在你结束这个连接时自动释放这个处理程序。

除了用来建立连接的代码，connect1.c 还用到了另外 3 个调用。

- MY_INIT()是一个初始化宏。它将把一个全局变量设置为指向你的程序名（你传给这个宏的参数值）供 MySQL 库在出错消息里使用。它还将调用 my_init()函数来完成一些初始设置工作。
- mysql_library_init()对 MySQL 客户端库进行初始化。应该在调用任何其他 mysql_xxx ()函数之前调用它。
- mysql_library_end()结束客户端库的使用并进行必要的清理工作。应该在用完客户端库之后调用它。

想试试 connect 程序？请按照本章前面介绍的步骤对这个客户程序进行编译和链接，然后运行。在 Unix 系统上，用下面的命令来运行程序：

```
% ./connect1
```

在 Unix 系统上，如果你的 shell 的搜索路径里不包括当前子目录（"."），这条命令里的"./"就不得省略。如果这个子目录在你的搜索路径里或者你使用的是 Windows 系统，你就可以省略命令名里的"./"，如下所示：

```
% connect1
```

如果 connect1 没有产生任何输出，就表明它已连接成功。如果不是这样，应该看到如下所示的消息：

```
% ./connect1
mysql_real_connect() failed
```

这样的输出消息只表明没有建立任何连接，但并没有告诉你原因何在。导致连接失败的常见原因之一是默认的连接参数（主机名、用户名等）不适当。如果真是如此，解决问题的办法之一是对初始化程序里的参数变量进行编辑，把它们改成能肯定让你访问到服务器的值，然后重编译程序。这么做的好处是至少可以让你建立起一个连接。但因为程序里包含着硬编码值，所以别人使用你的程序就不那么方便。它也不够安全，因为它暴露了你的口令。你或许会认为这些口令在你把程序编译成二进制可执行形式以后会变成隐藏的，但只要有人能运行 strings 工具去查看那些二进制代码，它们就会原形毕现。不仅如此，任何对源代码文件拥有读权限的人都可以轻而易举地看到这些口令。

上面这段论述揭示了 connect1 程序的两个明显不足。

- ❑ 出错消息让人们很难了解导致问题的具体原因。
- ❑ 无法让运行这个程序的用户灵活地设定连接参数，因为它们都已经硬编码在了源代码里。要是能让用户通过在选项文件里或是命令行上给出参数来“覆盖”它们就好了。

我们将在下一节解决这些问题。

7.3 出错消息和命令行选项的处理

我们的下一个客户程序 connect2 在功能上与 connect1 没有什么差异，如连接 MySQL 服务器、断开连接、退出。但 connect2 有两个重要的改进。

- ❑ 它在发生错误时能提供更多的信息。connect1 在遇到问题时只输出一条简短的消息，但我们可以把出错报告做得更好，因为 MySQL 客户端库里有几个函数可以返回关于出错原因的具体信息。
- ❑ 它可以让用户通过命令行或选项文件以选项的形式给出连接参数。

7

7.3.1 出错检查

让我们先来考虑出错检查问题。首先要强调这样一个观点：只要你调用的 MySQL 函数有可能失败，就必须检查它是否发生了错误。有许多程序设计图书都会“把出错检查部分留给读者们作为练习”，我个人认为这种回避的态度是因为出错检查工作确实很麻烦，让我们勇敢地面对它吧。不管怎么说，让 MySQL 客户程序测试出错条件并作出适当的处理是非常必要的。那些会返回各种状态值的客户端库函数之所以那么做都是有道理的，忽略它们是一种冒险行为。例如，如果某个函数在发生错误时返回一个指向某个数据结构的指针或者 NULL 值，你最好检查一下它的返回值以防万一。如果后面的程序代码需要的是一个指向某种合法数据结构的指针，而实际使用的却是 NULL 值，就会导致奇怪的结果，甚至导致整个程序崩溃。

不检查返回值往往会增加编程难度，MySQL 邮件列表上有相当一部分求助帖子都与此有关。常见的问题是“为什么我的程序会在发出这条语句时崩溃？”或“为什么我的查询命令没返回任何东西？”许多时候，出问题的程序要么是忘了在发出语句前检查连接是否已成功建立，要么是忘记了在检索查询结果之前检查服务器是否已成功地执行了语句。

千万不要想当然地认为 MySQL 客户端库里的函数在每次调用时都会成功。如果你忘了检查返回值，你将要一边忙着在程序里寻找一个难以跟踪的问题，或者向用户解释为什么程序会有奇怪的行为，或者两者兼顾。

根据返回值是指针还是整数，MySQL 客户端库函数用来返回值表明自己是否执行成功的办法有两种。

如果返回值是一个指针，那么，非 NULL 指针表示成功，NULL 表示失败。(NULL 在上下文中的含义是“C 语言中的 NULL 指针”，而不是“MySQL 数据库里的 NULL 数据列值。)

拿我们曾经使用过的 mysql_init()和 mysql_real_connect()函数来说，如果它们返回的是一个指向连接处理程序的指针，则表明调用成功；如果返回的是 NULL，则表明调用失败。

如果返回值是一个整数，那么，0 表示成功，非零值表示失败。注意：这个非零值并不是指某个特定的值，如-1。MySQL 客户端库里的函数在调用失败时的返回值是不可预料的，不能保证它是某个特定的数值。你也许见过有人用下面的代码测试 C API 函数 mysql_*XXX*()的返回值，但这种做法是错误的：

```
if (mysql_XXX () == -1)        /* this test is incorrect */
  fprintf (stderr, "something bad happened\n");
```

这个测试也许能够工作，也许不能工作。MySQL API 从没说过函数调用失败时的非零返回值是某个特定的值，它只能保证那个返回值不是零。正确的返回值测试代码应该是下面这样的：

```
if (mysql_XXX () != 0)         /* this test is correct */
  fprintf (stderr, "something bad happened\n");
```

下面这个测试也是正确的，它与上面的等价，但写起来稍微简单点：

```
if (mysql_XXX ())              /* this test is correct */
  fprintf (stderr, "something bad happened\n");
```

如果你阅读过 MySQL 本身的源代码，就会发现它使用第二种测试的情况要多一些。

并非每个 API 调用都会返回一个值。前面曾使用过的 mysql_close()就是一个没有返回值的函数。(它会调用失败吗？它调用失败了又怎样？你在这条连接上的工作反正已经完成了。)

如果某个 MySQL 客户端库里的函数调用失败了，可以利用这个 API 里的 3 个函数来查找原因。

- mysql_error()。返回一个包含出错信息的字符串。
- mysql_errno()。返回一个 MySQL 专用的数值型出错代码。
- mysql_sqlstate()。返回一个 SQLSTATE 代码。SQLSTATE 值是在 ANSI SQL 和 ODBC 标准里定义的，与数据库软件的品牌无关。

这几个函数的参数都是指向连接处理程序的指针。你应该在错误发生后立刻调用它们；如果你在调用它们之前又发出了另一个会返回状态信息的 API 调用，你从 mysql_error()、mysql_errno()或 mysql_sqlstate()获取的出错信息将来自后一个 API 调用。

一般说来，程序用户大都认为出错消息比出错代码更易于理解。因此，如果你想只返回一个值来报告错误的话，建议你返回一条出错消息。为完整起见，本章的例子在报告出错时将返回上述 3 个值。不过，在每一个可能发生错误的地方都写出 3 个函数调用很烦琐，所以我决定编写一个名为 print-error()的实用工具函数，把我们提供的出错消息和 MySQL 客户端库里的例程提供的出错值打印出来。换句话说，在每次测试错误时，将用不着再像下面这样去调用 mysql_errno()、mysql_error()和 mysql_sqlstate()函数了：

```
if (...some MySQL function fails...)
{
  fprintf (stderr, "...some error message...:\nError %u (%s): %s\n",
```

```
        mysql_errno (conn), mysql_sqlstate (conn), mysql_error (conn));
}
```

使用一个可以像下面这样被调用的实用工具函数来报告出错信息要简便得多：

```
if (...some MySQL function fails...)
{
  print_error (conn, "...some error message...");
}
```

print_error()函数将打印一条出错信息并调用 MySQL 的出错报告函数。print_error()调用比 fprintf()调用简单得多，它不仅更容易编写，还可以让程序代码更容易阅读。在此基础上，如果能把 print_error()编写得更细致，让它在 conn 是 NULL 时也能做些有实际意义的事情，我们甚至能够把它用在 mysql_init()调用失败或其他类似的场合里。出错报告调用将有统一的格式，不会再像以前那样混杂——有些是 fprintf()，有些是 print_error()。

有些读者可能会质疑："并不是每次报告时都必须调用所有的出错报告函数。你故意夸大了出错报告工作的繁琐，而目的不过是想让你的实用工具函数显得更有价值而已。再说了，你也用不着每次都去键入出错信息打印语句；你可以只写一次，然后复制并粘贴到需要用到它们的地方。"这些说法的确有一定的道理，但我也有我的理由。

- 复制加粘贴的办法，只对较短的代码比较方便。
- 不管你是否喜欢在每次报告错误时调用所有的出错报告函数，总是写出所有的出错报告代码迟早会让你觉得厌烦。你会自觉或不自觉地走一些"捷径"，从而导致你的出错报告工作前后不一致。把出错报告代码打包在一个便于调用的实用工具函数里能防微杜渐，使你的代码前后一致。
- 如果你后来又想修改出错信息的格式，只需修改一个地方肯定要比改整个程序更省事。或者，如果你后来又决定把出错信息写到一个日志文件里而不是写到 stderr 设备上去（或者同时写到这两个地方），只需修改 print_error()函数的做法将更省事。这种安排能够减少很多不必要的错误，而且能使你的代码前后一致。
- 如果用一个调试器来测试程序，你可以很方便地在出错报告函数里设置一个断点，使这个程序在每次检测到有错误发生时都能及时切换到调试器里去。

基于这些理由，本章后面的程序示例都将使用 print_error()函数来报告与 MySQL 有关的错误。下面的代码清单给出了 print_error()函数的定义，它具备前面讨论的优点：

```
static void
print_error (MYSQL *conn, char *message)
{
  fprintf (stderr, "%s\n", message);
  if (conn != NULL)
  {
    fprintf (stderr, "Error %u (%s): %s\n",
             mysql_errno (conn), mysql_sqlstate (conn), mysql_error (conn));
  }
}
```

connect2.c 程序里需要进行出错检查的部分与 connect.c 程序里的对应代码很相似，我们在这里使用了 print_error()函数，如下所示：

```
/* initialize connection handler */
```

7

```
conn = mysql_init (NULL);
if (conn == NULL)
{
  print_error (NULL, "mysql_init() failed (probably out of memory)");
  exit (1);
}

/* connect to server */
if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
    opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
{
  print_error (conn, "mysql_real_connect() failed");
  mysql_close (conn);
  exit (1);
}
```

出错检查逻辑的前提是 mysql_init()和 mysql_real_connect()函数在调用失败时都返回 NULL。请注意，如果 mysql_init()调用失败，我们将把 NULL 传递给 print_error()函数的第一个参数。这将使 print_error()函数不去调用 MySQL 的出错报告函数，因为我们不能在 mysql_init()执行失败的情况下仍想当然地认为传递给那些函数的连接处理程序还包含着有意义的信息。与此形成对照的是，当 mysql_real_connect()调用失败时，我们的确把连接处理程序传递给了 print_error()函数——这个处理程序虽然不会包含对应于合法连接的信息，却包含一些可以被递给出错报告类函数的诊断信息。你还可以把这个处理程序传递给 mysql_close()函数去释放 mysql_init()已经为它自动分配的各种内存。（切记，千万不要把这个处理程序传递给任何其他客户例程！因为它们大都只能在连接已经成功建立的前提下才能调用，你的程序可能会崩溃！）

本章里的其他示例程序都进行了出错检查，你在编写程序时也应如此。这看起来增加了编程工作量，但从长远来看，它反而是一种省力手段，因为你不必花费大量的时间去跟踪小问题。在第 8 章和第 9 章里，我将沿用这种出错检查策略。

7.3.2　实时获取连接参数

现在我们将让用户能够在运行时设定连接参数，而不是把连接参数的默认值硬编码在程序里。connect1 客户程序有一个显著的缺陷，即连接参数是直接写在源代码里的。不管需要改变它们当中的哪一个，你都不得不对源文件进行编辑并重新编译。这不太方便，要是你的程序还有其他使用者的话，事情将更麻烦。实时设定连接参数的常用方法之一是使用命令行选项。比如说，MySQL 发行版自带的很多程序都能接受两种形式的参数，如表 7-1 所示。

表　7-1

参　　数	长选项格式	短选项格式
主机名	--host=*host_name*	-h *host_name*
用户名	--user=*user_name*	-u *user_name*
口令	--password 或 --password=*your_pass*	-p 或 -p*your_pass*
端口号	--port=*port_num*	-P *port_num*
套接字名	--socket=*socket_name*	-S *socket_name*

为了与标准的 MySQL 客户程序保持一致，下一个客户程序 connect2，将采用同样的格式。这事做起来并不难，因为客户端库已经准备了一些选项处理功能。此外，connect2 还将具备从选项文件里提取信息的能力，这就使我们能够把连接参数放到~/.my.cnf 文件（即主目录里的.my.cnf 文件）或者某个全局选项文件里。这样，你就用不着在每次启动程序时都在命令行上设定那些选项了。客户端库能帮你寻找 MySQL 选项文件和从中提取相关值。只需在你的程序里增加几行代码，就可以让它支持选项文件机制——既然可以借助别人的智慧，你就用不着构思如何编写代码了。（F.2.2 节详细描述了选项文件的语法。）

在介绍 connect2 里选项处理如何工作之前，先来开发几个演示一般原理的程序。这些程序能够让你体会到选项处理工作其实是多么简单，它并不会增加连接 MySQL 服务器和处理查询命令的复杂性。

说明　MySQL 还提供并支持另外两种与建立服务器连接有关的选项，其一是用来指定连接协议（TCP/IP、Unix 套接字文件等）的--protocol 选项，其二是在 Windows 系统上给出用来建立共享内存连接的共享内存块名字的--shared-memory-base-name 选项。本章没有对这两个选项进行介绍，但收录在 sampdb 发行版本里的 protocol 程序演示了它们的用法，有兴趣的读者请自行参阅。

---

1. 访问选项文件的内容

从选项文件读出连接参数值的工作可以用 load_defaults()函数完成。load_defaults()函数将去寻找选项文件，分析它们的内容以找出你感兴趣的选项组、改写程序的参数向量（即 argv[]数组）。它会把来自那些选项组的信息以命令行选项的形式放在 argv[]数组的开头，使这些选项看起来就像是你在命令行上给出的那样。这样一来，你为分析各种命令行选项而编写的选项处理代码就可以“正常地”对那些来自选项文件的连接参数进行处理了。在 argv[]里，选项将出现在命令名的后面、其他参数的前面（注意，不是被追加在 argv[]的末尾）。所以，你在命令行上给出的任何连接参数将出现在 argv[]的后半段，因而能够覆盖由 load_defaults()添加进来的选项。

下面这个小程序的名字是 show_argv，它演示了 load_defaults()函数的使用方法，你将清楚地看到这个函数是如何改变 argv[]参数向量的：

```
/*
 * show_argv.c - show effect of load_defaults() on argument vector
 */

#include <my_global.h>
#include <my_sys.h>
#include <mysql.h>

static const char *client_groups[] = { "client", NULL };

int
main (int argc, char *argv[])
{
int i;

  printf ("Original argument vector:\n");
  for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);
```

```
  MY_INIT (argv[0]);
  load_defaults ("my", client_groups, &argc, &argv);

  printf ("Modified argument vector:\n");
  for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

  exit (0);
}
```

show_argv 程序里的选项文件处理代码由以下几个部分组成。

- client_groups[]。这个字符串数组存放着一些选项文件组的名字，而程序将去读取这些选项组里的选项。一般来说，客户端库至少应该把"client "放到这个数组里（它对应于[client]选项组），你可以在这个数组里列出任意多个选项组。这个数组的最后一个元素必须是 NULL，其作用是表明选项组名单到此为止。
- MY_INIT()。它是一个初始化宏。我们刚刚使用过它，但这里的重点是通过 MY_INIT()去调用 my_init()函数，以便完成 load_default()函数所要求的一些初始设置工作。
- load_defaults()。负责读取选项文件。它有 4 个参数，依次是：选项文件名中的前缀（这个前缀应该永远是"my"）、存放着你感兴趣的选项组名单的数组、程序的参数计数器和向量的地址。注意，因为 load_defaults()会改变程序的参数计数器和向量的值，所以你不能直接传递它们的值，而必须传递它们的地址。特别是 argv，虽然它本身已经是一个指针了，你也要把它作为&argv 传递，即该指针的地址。

为了让你看到 load_defaults()函数对，参数数组的作用效果，show_argv 程序将把它的参数打印两次：第一次打印的是你在命令行上给出的参数，第二次打印的则是调用 load_defaults()函数之后的参数数组。

如果你想看到 load_defaults()函数的工作情况，就必须保证在你的主目录里存在着一个名为.my.cnf 的文件，这个文件的[client]选项组里必须有几个选项。（在 Windows 系统上，你可以使用 C:\my.cnf 文件。）假设这个文件有如下所示的内容：

```
[client]
user=sampadm
password=secret
host=some_host
```

那么，运行 show_argv 程序将产生如下所示的输出：

```
% ./show_argv a b
Original argument vector:
arg 0: ./show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: ./show_argv
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: a
arg 5: b
```

在 show_argv 程序第二次打印出来的参数向量里，来自选项文件的值已经成为参数表的一部分了。你可能会在其中看到一些既不是你在命令行上给出的、也不是来自你~/.my.cnf文件的选项。如果真的如此，你应该能够在某个全局选项文件的[client]选项组里找到它们。之所以会出现这种情况，是因为 load_defaults()函数实际要去好几个位置读取选项文件。（F.2.2 节介绍了这些位置。）

需要使用 load_defaults()函数的客户程序几乎都会把"client"放到选项组名单里去（这是为了获得各选项文件对一般客户程序的基本设置），但你完全可以让你的选项文件处理代码从其他选项组里获取选项。比如说，如果你想让 show_argv 程序读取[client]和[show_argv]选项组里的选项，只需先找到 show_argv.c 里的下面这行代码：

```
const char *client_groups[] = { "client", NULL };
```

然后改写为如下所示的样子：

```
const char *client_groups[] = { "show_argv", "client", NULL };
```

再重新编译 show_argv 程序就能达到目的，新的 show_argv 程序将读取[client]和[show_argv]选项组里的选项。我们来验证一下，先在~/.my.cnf 文件里增加一个[show_argv]选项组：

```
[client]
user=sampadm
password=secret
host=some_host

[show_argv]
host=other_host
```

然后运行 show_argv 程序，你应该看到一个如下所示的输出，它与前面的不一样：

```
% ./show_argv a b
Original argument vector:
arg 0: ./show_argv
arg 1: a
arg 2: b
Modified argument vector:
arg 0: ./show_argv
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: --host=other_host
arg 5: a
arg 6: b
```

选项值在参数数组里的先后次序取决于它们在选项文件里的先后次序，而非选项组的名字在 client_groups[]数组里的先后次序。因此，在选项文件里，应该把客户程序专用的选项组放在[client]组的后面。这种安排的好处是：如果你在两个选项组里都设定了某个选项，供程序专用的值将覆盖掉[client]组里的普通值。你可以在刚才的例子里看到这一点：[client]和[show_argv]选项组都对 host 选项进行了设置，但因为[show_argv]选项组位于选项文件的最后，所以它给出的 host 值在参数向量里的出现位置将更靠后并因此而成为最终起效的选项值。

load_defaults()函数不会读取你通过环境变量给出的设置值。如果需要用到 MYSQL_TCP_PORT 或 MYSQL_UNIX_PORT 等环境变量的值，就必须通过 getenv()函数自行安排。本章的客户程序都没有用到环境变量。下面这段代码演示了如何检查标准的 MySQL 环境变量值：

```
extern char *getenv();
char *p;
int port_num = 0;
char *socket_name = NULL;

if ((p = getenv ("MYSQL_TCP_PORT")) != NULL)
  port_num = atoi (p);
if ((p = getenv ("MYSQL_UNIX_PORT")) != NULL)
  socket_name = p;
```

在标准的 MySQL 客户程序里，环境变量值的优先级要低于通过选项文件或命令行给出的值。如果你想在自己的程序里检查环境变量并与上述惯例保持一致，就应该把环境变量的检查工作安排在调用 load_defaults()函数或者处理命令行选项之前（而不是之后）。

load_defaults()与系统安全

在多用户系统上，有些实用工具（如 ps 程序）能够把任何进程（包括其他用户执行的）参数列表显示出来。那么，这类具备一定的进程嗅探功能的实用工具，会不会导致你用 load_defaults()函数从选项文件读出并放到参数列表里的口令泄密呢？不会，请不要为此担心，因为 ps 程序只显示 argv[]数组最初的内容。由 load_defaults()调用创建的口令参数将指向内存中专门为它分配的区域，这个区域并不是 argv[]向量的初始组成部分，所以 ps 程序看不到它。

不过，在命令行上给出的口令却会被 ps 程序显示出来，所以我建议大家不要在命令行上给出口令。为进一步降低风险，你可以让程序在开始执行之初就把口令从参数表里删除，其具体做法将在下一节介绍。

2. 处理命令行参数

有了 load_defaults()函数，我们就能把所有的连接参数都放到参数向量里，但现在需要处理这个向量。handle_options()函数就是为此准备的，它是 MySQL 客户端库的一部分，只要你链接了这个函数库，就可以使用它。

下面是客户端库中的选项处理例程的一些特点。

- 能够更精确地设定合法选项值的类型和取值范围。比如说，你不仅可以指定某个选项必须是一个整数值，还可以指定它必须是一个正整数且必须是 1024 的倍数。
- 增加了帮助功能。使调用一个标准库函数来打印帮助信息的工作更简便易行。你不必再专门编写一些代码来产生帮助信息了。
- 对标准的--no-defaults、--print-defaults、--defaults-file 和--defaults-extra-file 等选项有内建的支持机制。F.2.2 节将介绍这些选项。
- 支持一组标准的选项前缀（如--disable-、--enable-和--loose-），使布尔（开/关）型选项更便于使用。本章没有用到这些选项，请参考 F.2 节。

为了让大家对 MySQL 的选项处理机制有一个完整的认识，我们将在本节编写一个 show_opt 程序。这个程序将调用 load_defaults()函数来读取选项文件，修改参数向量，然后用 handle_options()函数对它们进行处理。

你可以利用 show_opt 程序来试验连接参数的各种设定方法（如通过选项文件或通过命令行），它会把最终用来连接 MySQL 服务器的参数值显示出来供你参考。show_opt 程序能帮助大家掌握下一个

客户程序 connect2 的工作情况，connect2 负责将这里的选项处理代码与我们前面开发的服务器连接代码结合起来。

show_opt 程序将用以下操作来告诉我们在参数处理的各个阶段发生了哪些事情。

(1) 把主机名、用户名、口令以及其他连接参数初始化为默认值。

(2) 把初始连接参数和参数向量的值打印出来。

(3) 调用 load_defaults()函数改写参数向量以反映选项文件的内容，然后把修改后的结果向量打印出来。

(4) 调用选项处理例程 handle_options()去处理参数向量，然后把连接参数值的最终结果和最后剩在参数向量里的东西打印出来。

下面是 show_opt 程序的源文件 show_opt.c 的内容，我们随后将对 show_opt 的工作流程做出说明。

```
/*
 * show_opt.c - demonstrate option processing with load_defaults()
 * and handle_options()
 */

#include <my_global.h>
#include <my_sys.h>
#include <mysql.h>
#include <my_getopt.h>

static char *opt_host_name = NULL;    /* server host (default=localhost) */
static char *opt_user_name = NULL;    /* username (default=login name) */
static char *opt_password = NULL;     /* password (default=none) */
static unsigned int opt_port_num = 0; /* port number (use built-in value) */
static char *opt_socket_name = NULL;  /* socket name (use built-in value) */

static const char *client_groups[] = { "client", NULL };

static struct my_option my_opts[] =   /* option information structures */
{
  {"help", '?', "Display this help and exit",
  NULL, NULL, NULL,
  GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
  {"host", 'h', "Host to connect to",
  (uchar **) &opt_host_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"password", 'p', "Password",
  (uchar **) &opt_password, NULL, NULL,
  GET_STR, OPT_ARG, 0, 0, 0, 0, 0, 0},
  {"port", 'P', "Port number",
  (uchar **) &opt_port_num, NULL, NULL,
  GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"socket", 'S', "Socket path",
  (uchar **) &opt_socket_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"user", 'u', "User name",
  (uchar **) &opt_user_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  { NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
```

```
};

static my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
  switch (optid)
  {
  case '?':
    my_print_help (my_opts);  /* print help message */
    exit (0);
  }
  return (0);
}

int
main (int argc, char *argv[])
{
int i;
int opt_err;

  printf ("Original connection parameters:\n");
  printf ("hostname: %s\n", opt_host_name ? opt_host_name : "(null)");
  printf ("username: %s\n", opt_user_name ? opt_user_name : "(null)");
  printf ("password: %s\n", opt_password ? opt_password : "(null)");
  printf ("port number: %u\n", opt_port_num);
  printf ("socket filename: %s\n",
          opt_socket_name ? opt_socket_name : "(null)");

  printf ("Original argument vector:\n");
  for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

  MY_INIT (argv[0]);
  load_defaults ("my", client_groups, &argc, &argv);

  printf ("Argument vector after calling load_defaults():\n");
  for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);

  if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option)))
    exit (opt_err);

  printf ("Connection parameters after calling handle_options():\n");
  printf ("hostname: %s\n", opt_host_name ? opt_host_name : "(null)");
  printf ("username: %s\n", opt_user_name ? opt_user_name : "(null)");
  printf ("password: %s\n", opt_password ? opt_password : "(null)");
  printf ("port number: %u\n", opt_port_num);
  printf ("socket filename: %s\n",
          opt_socket_name ? opt_socket_name : "(null)");

  printf ("Argument vector after calling handle_options():\n");
  for (i = 0; i < argc; i++)
    printf ("arg %d: %s\n", i, argv[i]);
```

```
  exit (0);
}
```

说明 show_opt.c 和本章稍后内容里的另外几个程序的源代码用到了与 MySQL 相关的数据结构中的 uchar**类型。在 MySQL 5.1.20 之前的版本里，你会发现 MySQL 头文件使用的是 gptr* 类型，它在你编译有关程序时会诱发警告。可以忽略这类警告消息。

show_opt.c 所实现的选项处理机制涉及以下几个方面，使用 MySQL 客户端库去处理命令行选项的任何程序都要按照这个套路来进行，你自己的程序也是如此。

(1) 除这里介绍的几个文件外，my_getopt.h 文件也要包括进来，它对 MySQL 选项处理例程的调用接口进行了定义。

(2) 定义 my_option 结构的数组，在 show_opt.c 文件里是 my_opts 数组。这个数组中的每一个 my_option 结构分别对应着该程序能够识别的一个选项，其内容是与该选项有关的各种信息，比如选项的短名字和长名字、它的默认值、它是一个数值还是一个字符串，等等。

(3) 在调用 load_defaults()函数读完选项文件并改写好参数向量之后，调用 handle_options()函数去处理那些选项。handle_options()函数的前两个参数分别是这个程序的参数计数值和向量。（注意：你传递的必须是这些变量的地址而不能是它们的值，这与 load_options()函数的情况很相似。）第三个参数指向 my_option 结构的数组。第四个参数是一个指针，它指向一个辅助函数。借助于 handle_options()函数和 my_option 结构客户端库能替你自动完成绝大多数的选项处理操作，但有些特殊动作是客户端库处理不了的，所以你的程序应该再定义一个辅助函数（show_opt.c 文件里的 get_one_option()函数）供 handle_options()调用。

my_option 结构对 MySQL 客户程序正确解读每一个选项所必需的信息的类型进行了定义：

```
struct my_option
{
  const char *name;               /* option's long name */
  int        id;                  /* option's short name or code */
  const char *comment;            /* option description for help message */
  uchar      **value;             /* pointer to variable to store value in */
  uchar      **u_max_value;       /* The user defined max variable value */
  struct st_typelib *typelib;     /* pointer to possible values (unused) */
  ulong      var_type;            /* option value's type */
  enum get_opt_arg_type arg_type; /* whether option value is required */
  longlong   def_value;           /* option's default value */
  longlong   min_value;           /* option's minimum allowable value */
  longlong   max_value;           /* option's maximum allowable value */
  longlong   sub_size;            /* amount to shift value by */
  long       block_size;          /* option value multiplier */
  void       *app_type;           /* reserved for application-specific use */
};
```

my_option 结构的各个成员的用法如下所示。

- ❑ name 是长选项名。它是以--name 形式的选项，但不包括前导的连字符。比如说，长选项--user 在 my_option 结构里将被写为"user"。
- ❑ id 是短（单个字母）选项名或一个与该选项相关联的代号（如果选项没有单字母名字）。比如

说，短选项-u 在 my_option 结构里将被写为'u'。如果选项只有长名字而没有对应的单字符名字，你可以指定一个代号作为它仅供内部使用短名字。代号必须是独一无二的，而且不能与现有的单字符名字相同。(为了满足后一条要求，你不妨把代号都设置得比 255（即单字符值的最大可取值）还要大。7.6 节有一个采用这一技巧的例子。)

- comment 是一个描述该选项用途的字符串。这是你希望出现在帮助信息里的文字。
- value 是一个泛型指针的地址，它被声明为一个 uchar**值。如果某个选项有一个参数，相应的 value 指针将指向你准备用来保存该参数的变量。在完成选项处理工作之后，查看该变量就可以知道选项的设置值到底是什么。value 指针所指向的变量的数据类型必须与 var_type 成员的值保持一致。如果某个选项没有任何参数，相应的 value 指针将被设置为 NULL。
- u_max_value 也是一个泛型指针，但只能由服务器程序使用。如果你正在编写的是客户程序，请把 u_max_value 设置为 NULL。
- typelib 目前尚未投入使用。在未来的 MySQL 版本里，你可以把选项的合法取值放到这个成员里，即要求用户给出的选项值必须与合法取值之一相匹配。
- var_type 指定在命令行上紧跟在选项名后面的那个值的类型。这些类型、含义和相应的 C 类型如表 7-2 所示。

表　7-2

var_type的可取值	含　　义	C　类　型
GET_NO_ARG	没有选项值	
GET_BOOL	布尔值	my_bool
GET_INT	整数值	int
GET_UINT	无符号整数值	unsigned int
GET_LONG	长整数值	long
GET_ULONG	无符号长整数值	unsigned long
GET_LL	长长整数值	long long
GET_ULL	无符号长长整数值	unsigned long long
GET_STR	字符串值	char*
GET_STR_ALLOC	字符串值	char*
GET_DISABLED	被禁用	
GET_ENUM	枚举值（目前未使用）	
GET_SET	集合值（目前未使用）	
GET_DOUBLE	双精度（浮点）值	double

从 MySQL 5.1.21 起，可使用 GET_DOUBLE。

GET_STR 和 GET_STR_ALLOC 的区别是：如果 var_type 取值为 GET_STR，选项变量将直接指向参数向量里的那个值；如果 var_type 取值为 GET_STR_ALLOC，系统将给该参数制作一个副本，而选项变量将指向这个副本。

GET_DISABLED 类型可以用来表明某个选项已不再有效，或是只在以某种特定方式编译程序时(比如说，启用了调试支持功能时）才有效。在 MySQL 源代码发行版本中的.mysql.cc 文件

里可以找到一个这样的例子。

- arg_type 指定选项名后面是否需要跟着一个选项值。它的可取值如表 7-3 所示。

表 7-3

arg_type的可取值	含 义
NO_ARG	选项名后面没有选项值
OPT_ARG	选项名后面的选项值允许省略
REQUIRED_ARG	选项名后面必须跟着一个选项值

如果 arg_type 的取值是 NO_ARG，var_type 成员就应该设置为 GET_NO_ARG。

- def_value，对于数值型的选项，如果没有在参数向量里明确地设定值，则使用这个值作为默认值。
- min_value，对于数值型的选项，它表示最小值。比这个值还小的值将被自动增大为这个值。0 表示没有最小值。
- max_value，对于数值型的选项，它表示最大值。比这个值还大的值将被自动减小为这个值。0 表示没有最大值。
- sub_size 表示选项值的偏移量。它用来调整数值型选项的取组范围：来自参数向量的选项值减去 sub_size 之后的结果才是系统内部使用的该选项的值。例如，如果命令行上给出的值的范围是 1 到 256，但程序内部实际使用的范围却是 0 到 255，sub_size 成员就将被设置为 1。
- block_size 是为数值型选项准备的。如果这个值不等于零，它就是一个单位块长度。如有必要，由用户给出的选项值将被自动向下取舍为这个长度的最近整数倍。比如说，如果选项值必须是偶数，把相应的单位块长度设置为 2 即可。handle_options()函数将把奇数值自动向下取舍为最接近的偶数。
- app_type 供应用程序使用。

每一个合法的选项在 my_opt 数组里都有一个对应的 my_option 结构。作为这个数组的结束标记，这个数组里的最后一个结构将是如下所示的样子：

```
{ NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
```

当你调用 handle_options()函数去处理参数向量时，它将跳过向量中的第一个参数（程序的名字）从第二个参数开始处理选项参数（即以短线开头的参数）。处理工作将一直进行到它到达向量的末尾或者遇到一个特殊的“终结者”参数（即两条短线--）为止。在遍历参数向量的过程中，每遇到一个选项，handle_options()就会调用辅助函数去进行相应的处理。handle_options()将向这个辅助函数传递 3 个参数：短选项值、一个指向对应于这个选项的 my_option 结构的指针、一个指向参数向量里紧跟在这个选项后面的那个参数的指针——如果这个选项不带选项值，这第 3 个参数将为 NULL。

当 handle_options()函数返回时，它会对参数的个数以及参数向量作出必要的调整，让参数表里只保留那些不是选项的参数。

下面是 show_opt 程序某次运行的输出结果（假设~/.my.cnf 文件的内容仍是前一小节里最后一次运行 show_argv 程序时的样子）：

```
% ./show_opt -h yet_another_host --user=bill x
Original connection parameters:
```

```
hostname: (null)
username: (null)
password: (null)
port number: 0
socket filename: (null)
Original argument vector:
arg 0: ./show_opt
arg 1: -h
arg 3: yet_another_host
arg 3: --user=bill
arg 4: x
Argument vector after calling load_defaults():
arg 0: ./show_opt
arg 1: --user=sampadm
arg 2: --password=secret
arg 3: --host=some_host
arg 4: -h
arg 5: yet_another_host
arg 6: --user=bill
arg 7: x
Connection parameters after calling handle_options():
hostname: yet_another_host
username: bill
password: secret
port number: 0
socket filename: (null)
Argument vector after calling handle_options():
arg 0: x
```

在上面的输出里，来自命令行的主机名将覆盖选项文件里设定的值，用户名和口令则仍使用选项文件的默认值。无论使用的是短选项格式（如-h yet_another_host）还是长选项格式（如--user=bill），handle_options()都正确地分析了这些选项。

辅助函数 get_one_option()是为了配合 handle_options()函数而编写的。它在 show_opt 程序里并没有多大的作用，它只对--help 或-?选项（此时，handle_options()传递给 get_one_option()的 optid 参数的值是'?'），做了一下处理。下面是辅助函数 get_one_option()的代码：

```
static my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
  switch (optid)
  {
  case '?':
    my_print_help (my_opts);  /* print help message */
    exit (0);
  }
  return (0);
}
```

my_print_help()是一个来自 MySQL 客户端库的例程，它的输入参数是一个选项名，它将根据这个选项名以及 my_option 数组里的注释字符串生成一条帮助信息。如果你想看看它的工作情况，可以试试下面这条命令：

```
% ./show_opt --help
```

根据具体情况，你还可以给 get_one_option()里的 switch()语句增加一些 case（我们稍后将在 connect2 程序中这样做）。比如说，你可以利用这个函数来处理口令选项。如果某个程序的口令选项是这样实现的，其用户在运行该程序时将既可以在命令行上给出口令，也可以不在命令行上给出口令而稍后再输入，就像选项信息结构里的 OPT_ARG 设置值所表明的那样。也就是说，你可以把口令选项写成--password 或--password=your_pass 的长选项形式，也可以写成-p 或-pyour_pass 的短选项形式。MySQL 客户程序通常允许你在命令行上省略口令值，等运行时再提示你输入它，这样做的好处是不会让别人看到你的口令。在本章后面的程序示例里，我们将用 get_one_option()函数来检查你是否在命令行上给出了口令值。如果你给出了口令值，我们就把它保存起来；否则，我们就设置一个标志，让程序在开始连接服务器之前提示你输入口令。

利用现在这个机会，你还可以对 show_opt.c 中的选项结构进行修改，看看对 show_opt 程序的行为到底有什么样的影响。比如说，如果你把--port 选项的 min_value、max_value、block_size 成员分别设置为 100、1 000、25，然后重新编译并运行这个程序，你将发现无法把端口号设置为从 100 到 1 000 这个范围以外的某个值，并且你给出的端口号值将自动舍入为与之最为接近的 25 的整数倍数。

选项处理例程还能自动完成对--no-defaults、--print-defaults、--defaults-file 和--defaults-extra-file 等选项的处理。请大家试试这些选项，调用 show_opt，看会发生什么事情。

7.3.3　给MySQL客户程序增加选项处理功能

7

现在。我们已经做好了编写 connect2.c 程序的全部准备工作。它将具备以下特点。

- 它可以连接 MySQL 服务器、断开连接、退出执行。这与 connect1.c 程序做的事情差不多，但 connect2.c 程序将使用我们在前面开发的 print_error()函数来报告错误。
- 它可以处理命令行或选项文件里的选项，使用的代码与 show_opt.c 文件里的类似，但 connect2.c 程序会在必要时提示用户输入口令。

下面是 connect2.c 程序的源代码：

```
/*
 * connect2.c - connect to MySQL server, using connection parameters
 * specified in an option file or on the command line
 */

#include <my_global.h>
#include <my_sys.h>
#include <m_string.h>   /* for strdup() */
#include <mysql.h>
#include <my_getopt.h>

static char *opt_host_name = NULL;     /* server host (default=localhost) */
static char *opt_user_name = NULL;     /* username (default=login name) */
static char *opt_password = NULL;      /* password (default=none) */
static unsigned int opt_port_num = 0; /* port number (use built-in value) */
static char *opt_socket_name = NULL;  /* socket name (use built-in value) */
static char *opt_db_name = NULL;       /* database name (default=none) */
static unsigned int opt_flags = 0;     /* connection flags (none) */

static int ask_password = 0;           /* whether to solicit password */
```

```
static MYSQL *conn;                   /* pointer to connection handler */

static const char *client_groups[] = { "client", NULL };

static struct my_option my_opts[] =   /* option information structures */
{
  {"help", '?', "Display this help and exit",
  NULL, NULL, NULL,
  GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
  {"host", 'h', "Host to connect to",
  (uchar **) &opt_host_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"password", 'p', "Password",
  (uchar **) &opt_password, NULL, NULL,
  GET_STR, OPT_ARG, 0, 0, 0, 0, 0, 0},
  {"port", 'P', "Port number",
  (uchar **) &opt_port_num, NULL, NULL,
  GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"socket", 'S', "Socket path",
  (uchar **) &opt_socket_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"user", 'u', "User name",
  (uchar **) &opt_user_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  { NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};

static void
print_error (MYSQL *conn, char *message)
{
  fprintf (stderr, "%s\n", message);
  if (conn != NULL)
  {
    fprintf (stderr, "Error %u (%s): %s\n",
             mysql_errno (conn), mysql_sqlstate (conn), mysql_error (conn));
  }
}

static my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
  switch (optid)
  {
  case '?':
    my_print_help (my_opts);  /* print help message */
    exit (0);
  case 'p':                   /* password */
    if (!argument)            /* no value given; solicit it later */
      ask_password = 1;
    else                      /* copy password, overwrite original */
    {
      opt_password = strdup (argument);
      if (opt_password == NULL)
```

```
      {
        print_error (NULL, "could not allocate password buffer");
        exit (1);
      }
      while (*argument)
        *argument++ = 'x';
      ask_password = 0;
    }
    break;
  }
  return (0);
}

int
main (int argc, char *argv[])
{
int opt_err;

  MY_INIT (argv[0]);
  load_defaults ("my", client_groups, &argc, &argv);

  if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option)))
    exit (opt_err);

  /* solicit password if necessary */
  if (ask_password)
    opt_password = get_tty_password (NULL);

  /* get database name if present on command line */
  if (argc > 0)
  {
    opt_db_name = argv[0];
    --argc; ++argv;
  }

  /* initialize client library */
  if (mysql_library_init (0, NULL, NULL))
  {
    print_error (NULL, "mysql_library_init() failed");
    exit (1);
  }

  /* initialize connection handler */
  conn = mysql_init (NULL);
  if (conn == NULL)
  {
    print_error (NULL, "mysql_init() failed (probably out of memory)");
    exit (1);
  }

  /* connect to server */
  if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
      opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
  {
```

7

```
    print_error (conn, "mysql_real_connect() failed");
    mysql_close (conn);
    exit (1);
  }

  /* ... issue statements and process results here ... */

  /* disconnect from server, terminate client library */
  mysql_close (conn);
  mysql_library_end ();
  exit (0);
}
```

与此前开发的 connect1 和 show_opt 程序相比，connect2 程序多了几种新本领。

- 你可以通过一个命令行参数来指定一个默认数据库。这与 MySQL 发行版本自带的标准客户程序是一致的。
- 如果输入参数向量里包含口令值，get_one_option()函数将在复制它后立刻改写那个原本。这将把时间窗口（别人有机会使用 ps 等系统状态查看程序看到命令行上给出的口令）压缩到最小。（注意：这只能把时间窗口压缩到最小，并不能彻底消除。在命令行上给出口令的做法仍有安防风险。）
- 如果你在命令行上只给出了口令选项而没有给出口令值，get_one_option()函数将设置一个标志让程序在运行时提示你输入口令，这项工作将由 main()函数在所有选项都被处理完毕之后调用 get_tty_password()函数去完成。get_tty_password()是客户端库里的一个实用工具函数，它在提示你输入口令时不会把口令回显在屏幕上。有些读者可能会问："为什么不用 getpass()函数来做这件事？"我的回答是："因为 getpass()函数并非在所有的系统上都能使用（比如说，它在 Windows 系统上就不能使用）"。get_tty_password()函数可以根据你使用的系统平台对自己作出调整，因而有着良好的跨系统移植性。

编译并链接 connect2 程序，然后运行：

```
% ./connect2
```

如果 connect2 程序没有产生任何输出（就像上面这样），则表示连接已成功建立。如果你看到的是下面这样的信息：

```
% ./connect2
mysql_real_connect() failed:
Error 1045 (28000): Access denied for user 'sampadm'@'localhost'
(using password: NO)
```

则表示连接没有建立起来，你现在可以知道它为什么没有建立起来了。在这个例子里，Access denied 意味着你还需要提供一些必要的连接参数。如果在运行 connect1 程序时遇到这种情况，你只能通过编辑源文件、再重新编译它的办法来解决问题。connect2 程序将根据你在命令行或选项文件里给出的选项去连接 MySQL 服务器。为简明起见，以下讨论将假设你没有使用任何选项文件。如果你在运行 connect2 程序时没有给出任何参数，它将尝试连接 localhost 主机并把你的 Unix 登录名和空口令发送到服务器。如果你使用了如下所示的命令行来运行 connect2 程序，它将提示你输入一个口令（因为在-p 选项的后面没有紧跟一个口令值），尝试连接 some_host 主机，并把用户名 some_user 以及你输入的口令传递给服务器：

```
% ./connect2 -h some_host -p -u some_user some_db
```

connect2 程序还将把数据库名 some_db 传递给 mysql_real_connect()函数，如果连接成功，它将成为默认的数据库。如果你在事先还准备了一个选项文件，connect2 程序还将对该文件的内容进行处理并对连接参数进行相应的修改。

让我们稍微后退一步，总结一下已经取得的成果。我们让 connect2 程序具备的功能是每一个 MySQL 客户程序都应该具备的：使用适当的参数去连接服务器。它在报告出错方面做的也很不错——当然，你只有在它未能成功连接服务器时才能体会到这一点。我们现在有一个可以用来编写各种 MySQL 客户程序的基础框架。如果你想编写一个新的 MySQL 客户程序，只要按以下步骤进行就可以了。

(1) 复制一份 connect2.c 文件的副本。

(2) 如果新程序支持的选项比我们在 connect2.c 程序里实现的标准选项多，就把新增加的选项添加到 my_opt 数组里并对 connect2.c 程序里的选项处理循环作必要的修改。

(3) 把新程序特有的代码添加到用来连接服务器和与服务器断开连接的两个调用之间。

OK 了，新程序的所有实际动作将发生在 mysql_real_connect()和 mysql_close()调用之间。有了一个像 connect2.c 程序这样可以重复利用的编程框架，意味着你可以把你的注意力集中于你真正感兴趣的东西——如何访问数据库里的内容。

7.4 处理 SQL 语句

我们连接 MySQL 服务器的目的是为了在连接打开时与它通信并交流信息。本小节的学习重点是怎样与服务器通信来处理语句。每个语句都要通过以下几个步骤才能得到妥善的处理。

(1) 构造语句。完成这一步骤的方法取决于语句本身的内容，具体地说，就是看它是否包含二进制数据。

(2) 把语句发送给服务器去执行。服务器将执行这个语句并生成一个结果。

(3) 对结果进行处理。具体做法要由你所发出的语句的类型来决定。比如说，SELECT 语句通常都会返回一些数据行让你处理，INSERT 语句却不会这样。

MySQL 客户端库收录了两组语句执行例程。第一组例程把每条语句当做一个字符串发送给服务器，而由服务器返回的所有数据列将按字符串格式进行检索。第二组例程使用了一种二进制协议，能按非字符串数据值的原始格式进行发送和接收，不需要对它们来回进行字符串格式转换。

在这一节里，我们将只讨论对 SQL 语句的第一种处理办法。采用二进制协议的处理办法将在 7.9 节讨论。

在构造语句时，将用哪个函数把它们发往服务器也是要考虑的因素之一。最常用的语句发送例程是 mysql_real_query()。这个例程把语句当做一个计数字符串（一个字符串加上一个长度值）来对待，你必须计算出语句字符串的长度并把这个长度值随字符串一起传递给 mysql_real_query()。因为语句将被视为一个计数字符串而不是一个以 NULL 结尾的字符串，所以它可以容纳包括二进制数据和 NULL 在内的任何东西。

语句也可以用 mysql_query()函数发出，这个函数用起来要简便一些，但它对语句字符串的限制就要多一些。你传递给 mysql_query()函数的语句必须是一个以 NULL 结尾的字符串，这就意味着语句本身不得包含 NULL 字节，因为 NULL 字节将导致语句被解释得比实际短。一般说来，如果语句可能包含任意二进制数据，就有可能包含 NULL，你也就不应该使用 mysql_query()。但从另一方面讲，如果语句肯定是一个以 NULL 结尾的字符串，你就可以利用 C 语言标准函数库中的字符串函数（如

strcpy()和sprintf()等）来构造它们，这些函数应该是你已运用得非常熟练的了。

在构造语句时，还必须考虑到特殊字符的转义问题。如果你正在构造的语句包含有二进制数据或者引号、反斜线等具有特殊含义的字符，就必须对它们进行转义处理。7.4.7节的第1小节将讨论这个问题。

下面是查询处理机制的一个简单框架：

```
if (mysql_query (conn, stmt_str) != 0)
{
  /* failure; report error */
}
else
{
  /* success; find out what effect the statement had */
}
```

如果语句执行成功，mysql_query()和mysql_real_query()都将返回零；否则，它们都将返回一个非零值。所谓“成功的”语句指的是MySQL服务器认为没有语法错误、能够执行的语句，与语句的执行效果没有任何关系。比如说，成功的SELECT查询不见得会返回一些数据行，而成功的DELETE语句也不见得真的删除了某些数据行。语句的实际执行效果要用其他手段来检测。

语句失败的原因有好几种。下面是一些比较常见的失败原因。

- 语句本身有语法错误。
- 语句在语义上有错误——比如语句里用到了一个其实并不存在的数据表。
- 语句将要访问某个数据表，但你却没有足够的权限。

我们可以把语句粗略地划分为两大类：一类是修改数据行的，另一类则是会返回一个结果集（一组数据行）的。INSERT、DELETE、UPDATE等语句修改数据行并告知你有多少个数据行受到了它们的影响。

SELECT和SHOW等语句返回一个结果集。在MySQL C API里，这类语句所返回的结果集将被表示为MYSQL_RES数据类型。这种数据类型其实是一个结构，它里面容纳着数据行的数据值以及关于这些数据值的元数据，如数据列的名字和数据值的长度等。结果集允许为空，即允许结果集里的数据行个数是零。

7.4.1　处理修改数据行的语句

要想对一个修改数据行的语句进行处理，首先要通过mysql_query()或mysql_real_query()调用把它发送给服务器去执行。如果语句成功，你就可以调用mysql_affected_rows()函数去查知它实际插入、删除、修改了多少个数据行了。

下面是一个用来对修改数据行的语句进行处理的示例：

```
if (mysql_query (conn, "INSERT INTO my_tbl SET name = 'My Name'") != 0)
{
  print_error (conn, "INSERT statement failed");
}
else
{
  printf ("INSERT statement succeeded; number of rows affected: %lu\n",
          (unsigned long) mysql_affected_rows (conn));
}
```

请注意，这段代码是把 mysql_affected_rows()函数的返回值强制转换为一个 unsigned long 值之后才把它打印出来的。mysql_affected_rows()函数的返回值是一个 my_ulonglong 类型的值，但这种类型的值在某些系统上无法直接打印出来。作为一种通用的解决方案，你需要把返回值强制转换为 unsigned long 类型并使用%lu 打印格式符。这一解决方案也适用于其他一些会返回 my_ulonglong 值的函数，如 mysql_num_rows()和 mysql_insert_id()等。如果想让你编写的客户程序具备跨系统的可移植性，千万要记住这个技巧。

mysql_affected_rows()的返回值能告诉你有多少数据行受到了语句的影响，而这个“影响”的具体含义还要取决于语句的类型。对 INSERT、REPLACE、DELETE 语句来说，这个数字是指它们插入、替换、删除了多少个数据行。对 UPDATE 语句来说，这个数字指的是它实际修改了多少个数据行。如果数据行在修改前后内容没有发生变化，MySQL 将认为它没有被修改。也就是说，即使某个数据行符合 UPDATE 语句的 WHERE 子句所给出的选取条件，也并非一定会发生改变。

UPDATE 语句的“受影响数据行”的这种含义时不时地会引起争论，有些人希望它的含义是“匹配数据行”，也就是符合更新筛选条件的数据行的总数，哪怕更新操作实际上并没有改变它们的值。如果某个应用程序确实需要用到后一种含义，可以在连接服务器时通过标志参数里的把 CLIENT_FOUND_ROWS 值传递给 mysql_real_connect()函数的办法要求 MySQL 服务器提供这种行为。

7.4.2 处理有结果集的语句

有些语句是会返回一个结果集的。在你用 mysql_query()或 mysql_real_query()发出这类语句之后，它们从数据库里检索出来的数据将被返回为一个结果集供你做进一步的处理。注意，在 MySQL 里，能返回数据行的语句并非仅有 SELECT 这一条，SHOW、DESCRIBE、EXPLAIN、CHECK TABLE 等语句也会返回结果集。对于以上这些语句，你在发出语句后通常还需要对它们返回的结果集做进一步的处理。

结果集的处理工作涉及以下几个步骤。

(1) 调用 mysql_store_result()或 mysql_use_result()函数去生成一个结果集。这两个函数在调用成功时都将返回一个 MYSQL_RES 指针，如果执行失败，则都将返回 NULL。我们稍后将会对这两个函数之间的区别和它们各自的适用场合进行介绍。但就眼下来说，我们将以 mysql_store_result()为例展开讨论，这个函数会立刻从服务器检索出数据行并把它们缓存在客户端的内存里。

(2) 调用 mysql_fetch_row()函数依次取回结果集里的各个数据行。这个函数在调用成功时将返回一个 MYSQL_ROW 值，如果已经到达结果集里的最后一个数据行，则将返回 NULL。MYSQL_ROW 值其实是一个字符串数组指针，数组中的字符串代表着数据行中的各个数据列的值。如何处理这些数据行要由应用程序的用途来决定。你可以简单地把它们都打印出来，也可以对它们进行统计分析，或者做一些其他的处理。

(3) 处理完结果集之后，调用 mysql_free_result()函数去释放它占用的内存资源。如果你忘了做这件事，你的应用程序就会有内存泄漏。如果应用程序的运行时间比较长，就要特别注意及时释放那些不再会被用到的结果集；否则，你的系统就会因资源消耗量持续增长而变得越来越慢。

下面是一个用来对有结果集的语句进行处理的示例：

```
MYSQL_RES *res_set;

if (mysql_query (conn, "SHOW TABLES FROM sampdb") != 0)
```

```
  print_error (conn, "mysql_query() failed");
else
{
  res_set = mysql_store_result (conn);  /* generate result set */
  if (res_set == NULL)
    print_error (conn, "mysql_store_result() failed");
  else
  {
    /* process result set, and then deallocate it */
    process_result_set (conn, res_set);
    mysql_free_result (res_set);
  }
}
```

这段代码把结果集的处理细节都隐藏在了还未定义的 process_result_set()函数里。一般来说，结果集的处理工作都是以一个如下所示的循环为基础的：

```
MYSQL_ROW row;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
  /* do something with row contents */
}
```

mysql_fetch_row()将返回一个 MYSQL_ROW 值，它是一个数组指针。如果你把这个返回值赋值给一个名为 row 的变量，就可以利用 row[*i*]语法来访问其中的各个元素，*i* 的最小值是 0，最大值比该数据行里的数据列个数少 1。MYSQL_ROW 数据类型有以下几个值得注意的要点。

- MYSQL_ROW 是指针类型，所以你在定义这种类型的变量时应该把它写成 MYSQL_ROW row 而不能写成 MYSQL_ROW *row。
- 在 MYSQL_ROW 数组里，一切数据类型，包括数值类型，都将被返回为字符串。如果你想把某个值用作数字，就必须亲自对它进行类型转换。
- MYSQL_ROW 数组里的字符串都是以 NULL 结尾的。但是，因为那些用来存放二进制数据的数据列里可能包含 NULL，所以你不能把这个数组里的值当做以 NULL 结尾的字符串来对待。你必须通过数据列的长度来了解数据列的值。（数据列长度的确定办法将在 7.4.6 节中介绍。）
- 在 MYSQL_ROW 数组里，数据库里的 NULL 值将被表示为一个 NULL 指针。如果数据列没有被定义为 NOT NULL，你就应该在程序里检查该数据列里的值是否为 NULL，否则，你的程序就会因试图对 NULL 指针进行解引用而崩溃。

如何处理数据行要由应用程序的具体用途来决定。作为演示，这里的示例程序将只把结果集里的每个数据行打印出来，数据列值之间用制表符隔开。要想做到这一点，就必须知道数据行由多少个数据列构成，这可以用另一个客户端库函数 mysql_num_fields()查出来。

下面是 process_result_set()函数的源代码：

```
void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
MYSQL_ROW    row;
unsigned int i;

  while ((row = mysql_fetch_row (res_set)) != NULL)
```

```
    {
      for (i = 0; i < mysql_num_fields (res_set); i++)
      {
        if (i > 0)
          fputc ('\t', stdout);
        printf ("%s", row[i] != NULL ? row[i] : "NULL");
      }
      fputc ('\n', stdout);
    }
    if (mysql_errno (conn) != 0)
      print_error (conn, "mysql_fetch_row() failed");
    else
      printf ("Number of rows returned: %lu\n",
              (unsigned long) mysql_num_rows (res_set));
  }
```

process_result_set()函数将依次打印各数据行的内容，数据列值之间用制表符分隔（NULL 值将被打印为单词“NULL”）；然后再把检索到的数据行的个数打印出来，这个计数值是通过调用 mysql_num_rows()函数得到的。类似于 mysql_affected_rows()，mysql_num_rows()函数的返回值也是一个 m_ulonglong 值，所以我们得先把它强制转换为 unsigned long 类型、再用%lu 格式符打印它。但同时要注意：mysql_affected_rows()函数的参数是一个连接处理程序，而 mysql_num_rows()函数的参数却是一个结果集指针。

循环结束后的代码包含一个错误检测，作出这种预防性安排的的理由是：如果结果集是用 mysql_store_result()创建的，那么 mysql_fetch_row()的 NULL 返回值将永远表示“已经到达结果集的末尾”；可如果结果集是用 mysql_use_result()创建的，那么 mysql_fetch_row()的 NULL 返回值就有“已经到达结果集的末尾”和“发生错误”两种含义。因为 process_result_set()并不知道自己的调用者是用 mysql_store_result()还是用 mysql_use_result()去创建结果集的；所以，为了让它在这两种情况下都能正确地检测到出错情况，我们给它加上了这个出错情况测试。

在打印数据列值时，process_result_set()函数的这一版本采用了最简单也最粗糙的做法。假设执行的是如下所示的查询：

```
SELECT last_name, first_name, city, state FROM president
ORDER BY last_name, first_name
```

你将看到下面的输出，它可算不上整齐：

```
Adams   John    Braintree   MA
Adams   John Quincy Braintree   MA
Arthur  Chester A.  Fairfield   VT
Buchanan    James   Mercersburg PA
Bush    George H.W. Milton  MA
Bush    George W.   New Haven   CT
Carter  James E.    Plains  GA
...
```

如果能给这份输出加上数据列名称作为列标题，再把数据沿纵向对齐，效果就会好得多。这就需要我们知道各数据列的名字和各数据列里最宽的值。这些信息是存在的，但它们不是数据列的数据值的一部分，而是结果集的元数据（即关于数据的数据）的组成部分。等介绍完语句处理程序之后，我们将在 7.4.6 节编写一个更整齐美观的打印的格式程序。

如何打印二进制数据

包含二进制数据的数据列值可能包含 NULL 字节，这种数据是无法用 printf()函数的%s 格式说明符打印的。printf()函数把 NULL 字节视为字符串的结束标记，所以它只能把数据列值中的第一个 NULL 字节之前的内容打印出来。因此，要想把二进制数据完整地打印出来，就必须使用接受数据列长度参数的函数，如 fwrite()函数。

7.4.3 一个通用的语句处理程序

前两个小节里的语句处理示例的代码都是根据语句是否会返回一个结果集而编写的。这种做法能够奏效的原因是我们把语句硬编码在了程序代码里：我们在示例中分别使用了一条不会返回结果集的 INSERT 语句和一条会返回结果集的 SHOW TABLES 语句。

可是，这种事先知道将要对哪一种查询进行处理的好事不会总让你遇到。比如说，如果你要执行的语句是从键盘或者某个文件读入的，那么这条语句的内容可能是任意的，你不仅很难在事先知道它是否会返回数据行，就连它是否是一条合法的 SQL 语句都成问题。你该怎么办？你肯定不想通过语法分析它们到底是哪一种 SQL 语句，这不像看上去那么简单。例如，如果只检查语句的第一个单词是不是 SELECT，就无法对付下面这种以注释开头的语句：

```
/* comment */ SELECT ...
```

还好，有了 MySQL C API 的帮忙，你不必提前知道语句类型就能对它作出正确的处理。下面，我们将利用 MySQL C API 来编写一个通用的语句处理程序，它能对各种 SQL 语句作出正确的处理，而不管它是否会返回一个结果集，也不管它的执行是否成功。在开始编写这个处理程序的代码之前，我们先把它的工作流程概括为如下。

(1) 发出语句。如果它执行失败，则就此结束。

(2) 如果语句成功，调用 mysql_store_result()函数从服务器检索出有关的数据行并创建一个结果集。

(3) 如果 mysql_store_result()调用成功，语句将返回一个结果集。调用 mysql_fetch_row()函数对结果集里的数据行进行处理直到它返回 NULL 为止，然后释放这个结果集。

(4) 如果 mysql_store_result()调用失败，其原因可能是语句根本不会返回一个结果集，也可能是语句会返回一个结果集，但在试图创建结果集时发生了错误。这两种情况可以利用 mysql_field_count()函数来区别：把连接处理程序传递给这个函数，然后检查它的返回值。

- 如果 mysql_field_count()返回的是 0，说明这个语句一个数据列也没有返回，也就是没有结果集。（这同时也表明语句是 INSERT、DELETE、UPDATE 等语句中某一个。）
- 如果 mysql_field_count()返回的是一个非零值，就表明发生了错误，因为该语句应该返回一个结果集。发生这类错误的原因有很多，比如因结果集尺寸过大而导致内存分配失败，或者客户/服务器之间的网络连接在提取数据行时中断。

下面这个函数能够处理任何语句，它的参数有两个：一个是连接处理程序，另一个是以 NULL 字节结尾的语句字符串。

```
void
process_statement (MYSQL *conn, char *stmt_str)
{
```

```
MYSQL_RES *res_set;

  if (mysql_query (conn, stmt_str) != 0)  /* the statement failed */
  {
    print_error (conn, "Could not execute statement");
    return;
  }

  /* the statement succeeded; determine whether it returned data */
  res_set = mysql_store_result (conn);
  if (res_set)      /* a result set was returned */
  {
    /* process rows and then free the result set */
    process_result_set (conn, res_set);
    mysql_free_result (res_set);
  }
  else              /* no result set was returned */
  {
    /*
     * does the lack of a result set mean that the statement didn't
     * return one, or that it should have but an error occurred?
     */
    if (mysql_field_count (conn) == 0)
    {
      /*
       * statement generated no result set (it was not a SELECT,
       * SHOW, DESCRIBE, etc.); just report rows-affected value.
       */
      printf ("Number of rows affected: %lu\n",
              (unsigned long) mysql_affected_rows (conn));
    }
    else  /* an error occurred */
    {
      print_error (conn, "Could not retrieve result set");
    }
  }
}
```

7.4.4　另一种语句处理方案

上一节里的 process_query()函数有下面 3 个特点。

- 它使用 mysql_query()来发出语句。
- 它使用 mysql_store_result()来检索结果集。
- 在没有获得结果集时，它利用 mysql_field_count()来判断其原因是语句执行出错还是它根本就不会返回结果集。

如果对这 3 个方面加以改变，就能得到另一种查询处理方案。

- 用一个计数语句字符串和 mysql_real_query()代替那个以 NULL 字节结尾的字符串和 mysql_query()。
- 用 mysql_use_result()代替 mysql_store_result()去创建结果集。
- 用 mysql_error()或 mysql_errno()代替 mysql_field_count()去区分“执行出错”和“没

有结果集可供返回”这两种情况。

你可以按这 3 种情况的任意组合对 process_query()函数进行修改。下面这个 process_real_query()函数就是我对 process_query()函数的上述 3 个方面全部替换后得到的：

```
void
process_real_statement (MYSQL *conn, char *stmt_str, unsigned int len)
{
MYSQL_RES *res_set;

  if (mysql_real_query (conn, stmt_str, len) != 0) /* the statement failed */
  {
    print_error (conn, "Could not execute statement");
    return;
  }

  /* the statement succeeded; determine whether it returned data */
  res_set = mysql_use_result (conn);
  if (res_set)      /* a result set was returned */
  {
    /* process rows and then free the result set */
    process_result_set (conn, res_set);
    mysql_free_result (res_set);
  }
  else              /* no result set was returned */
  {
    /*
     * does the lack of a result set mean that the statement didn't
     * return one, or that it should have but an error occurred?
     */
    if (mysql_errno (conn) == 0)
    {
      /*
       * statement generated no result set (it was not a SELECT,
       * SHOW, DESCRIBE, etc.); just report rows-affected value.
       */
      printf ("Number of rows affected: %lu\n",
              (unsigned long) mysql_affected_rows (conn));
    }
    else  /* an error occurred */
    {
      print_error (conn, "Could not retrieve result set");
    }
  }
}
```

7.4.5　mysql_store_result()与mysql_use_result()函数的对比

mysql_store_result()与 mysql_use_result()函数的共同点有两个：一是都以一个连接处理程序作为参数，二是都会返回一个结果集。它们之间的差异却远多于此。最本质的区别在于它们用来从服务器检索结果集数据行的方式不一样：mysql_store_result()会在你调用它时立刻把所有的数据行全都检索出来；mysql_use_result()则只完成对检索的初始化工作，它本身并不取回任何数据

行。正是这一区别导致了这两个函数在其他方面的种种差异。为了让大家能够在编写 MySQL 客户程序时在这两个函数之间作出最适当的选择，本节将对这两个函数进行比较。

在从服务器检索结果集时，mysql_store_result()会取回数据行，为它们分配内存，然后把它们保存在客户端。此后的 mysql_fetch_row()调用永远不会返回出错消息，因为它们只是从一个已经包含结果集的数据结构里提取出一个数据行而已。因此，mysql_fetch_row()函数的 NULL 返回值永远是"已经到达结果集的末尾"的含义。

再看 mysql_use_result()，它本身不会去检索任何数据行。它只是逐行完成对检索的初始化工作，真正取回各数据行的工作还要由你通过调用 mysql_fetch_row()去完成。因此，虽然 mysql_fetch_row()调用的 NULL 返回值通常表示"已经到达结果集的末尾"，但也有可能表示"与服务器的通信出现了问题"。你可以利用 mysql_error()或 mysql_errno()调用来区分这两种情况。

因为要把结果集完整地保存在客户端，所以，与 mysql_use_result()相比，mysql_store_result()消耗的内存和处理需求更多，因分配内存和创建数据结构而导致的开销也更大。过于巨大的结果集会给客户端带来内存消耗殆尽的风险。因此，当你要检索的结果集包含多个数据行时，就应该使用 mysql_use_result()。

mysql_use_result()因为每次只取回一个数据行进行处理，所以它对内存的要求很低。同时，因为不必为创建结果集而建立各种复杂的数据结构，它的内存分配工作也将完成得更快。但从另一个角度看，mysql_use_result()加重了服务器的负担，服务器必须把结果集里的数据行一直保存到客户程序检索它们时为止。因此，mysql_use_result()不适合用在以下几种客户程序里。

- 根据用户请求逐个遍历各有关数据行的交互式客户程序。(你肯定不想让服务器因为用户去喝咖啡了而一直等着发送下一个数据行吧？)
- 在两次数据行检索操作之间需要进行大量处理的客户程序。

在上述两种情况里，客户程序都无法迅速地把结果集里的数据行全部检索完毕。这对服务器和其他客户程序都有很大的负面影响，特别是，如果你正在使用 MyISAM 这种有数据表锁定功能的存储引擎，因为你从中检索数据的那些数据表在你查询期间将一直处于读操作锁定状态，试图修改这些数据表的其他客户程序都将因此而被阻塞。

虽说 mysql_store_result()会消耗较多的内存，但让你能对整个结果集进行访问却是一件好事。因为结果集里的数据行都存放在客户主机里，所以你能随时访问它们；你可以利用 mysql_data_seek()、mysql_row_seek()、mysql_row_tell()等函数按任意顺序去访问数据行。可如果你使用的是 mysql_use_result()，就只能按 mysql_fetch_row()取回数据行时的顺序访问数据行。如果你想按任意顺序而不是按它们从服务器被取回的顺序去处理数据行，就必须使用 mysql_store_result()。比如说，如果你想让应用程序允许用户跳跃地前后浏览你用某个查询选取出来的数据行，mysql_store_result()就应该是你的首选。

mysql_store_result()还能让你访问一些你在使用 mysql_use_result()时访问不到的数据列信息。比如说，你可以通过 mysql_num_rows()调用查知结果集总共包含多少个数据行；可以从 MYSQL_FIELD 数据列信息结构的 max_width 成员查知各数据列的数据最大宽度。可如果使用的是 mysql_use_result()，mysql_num_rows()将只有在数据行全部取回之后才会返回正确的计数值，类似地，max_width 成员的值也只有在数据行全部取回之后才能正确地计算出来，在此之前将不可用。

因为 mysql_use_result()做的事情比 mysql_store_result()少，所以它必须遵守一条 mysql_store_result()不必遵守的规定——客户程序必须通过 mysql_fetch_row()调用取回结果

集里的每个数据行。如果你在发出另一条语句之前忘了这么做，当前结果集里尚未来得及取回的数据行就将混杂在下一个语句的结果集里，而你则会看到一条“out of sync”（数据不同步）出错信息。（如果在发出第二个语句之前调用了 mysql_free_result() 函数，就可以避免出现这一问题。mysql_free_result()将替你取回并删除当前结果集里尚未被取回的数据行。）这条规定还隐含着这样一层含义：如果使用的是 mysql_use_result()，那你每次只能使用一个结果集进行工作。

mysql_store_result()不会导致数据不同步问题的发生，因为当这个函数返回时，服务器上就不会再有尚未被取回的数据行了。事实上，如果使用的是 mysql_store_result()，根本用不着显式调用 mysql_fetch_row()函数。不过，如果你感兴趣的只是结果集是否为空而不是结果集里有什么样的数据，这个函数还是有些用处的。比如说，你想知道数据库里是否存在一个名为 mytbl 的数据表，于是可以发出一个如下所示的语句：

```
SHOW TABLES LIKE 'mytbl'
```

如果在调用 mysql_store_result()之后，mysql_num_rows()返回的是一个非零值，就说明这个数据表是存在的。就这个例子而言，不需要调用 mysql_fetch_row()。

虽说应该及时调用 mysql_free_result()函数去释放用 mysql_store_result()生成的结果集，但并非必须在发出下一个语句之前这样做。这意味着你可以同时生成并使用多个结果集进行工作，这与 mysql_use_result()要求你每次只能使用一个结果集的规定形成了鲜明的对照。

如果想向用户提供最大限度的灵活性，可以让用户去选择结果集的处理方案。mysql 和 mysqldump 就是两个很好的例子：在默认的情况下，它们将使用 mysql_store_result()；可如果你在命令行上给出了--quick 选项，它们就将使用 mysql_use_result()。

7.4.6　使用结果集元数据

结果集不仅包含从数据库里检索出来的数据，还包含关于这些数据的信息，即所谓的结果集“元数据”，如下所示。

- 结果集里的数据行个数和数据列个数。只需调用 mysql_num_rows()和 mysql_num_fields()函数就能把它们查出来。
- 当前数据行里各数据列值的长度。只需调用 mysql_fetch_lengths()函数就能把它们查出来。
- 关于各数据列的信息，如数据列的名字和类型、各数据列的数据最大宽度、结果集里的数据列所在的数据表，等等。这些信息都保存在 MYSQL_FIELD 结构里，你可以调用 mysql_fetch_field()函数去获取。在线资源附录 G 详细介绍了 MYSQL_FIELD 结构以及所有供你用来访问数据列信息的函数。

元数据是否有效部分取决于你采用的结果集处理方案。7.4.5 节介绍过，如果要用到数据行计数值或者各数据列的数据最大宽度，就必须使用 mysql_store_result()而不是 mysql_use_ result()去创建结果集。

结果集元数据能帮你决定如何处理结果集数据。

- 如果想生成一份带有列标题并沿纵向整齐排列的输出报告，就需要用到各数据列的名字和它们的数据宽度。
- 数据列计数值可以让我们知道依次处理全部数据列值的循环要执行多少次才能处理完一个数据行。

- 在需要根据结果集中的数据行和数据列个数来为数据结构分配内存时，可以使用数据行计数值和数据列计数值。
- 可以利用结果集元数据来确定数据列的数据类型，由此你可以清楚地掌握哪些数据列是数值类型的，它们是否包含二进制数据，等等。

在 7.4.2 节里，我们编写的 process_result_set()函数将把结果集数据的数据列以制表符分隔的格式打印出来。这个程序很有用（比如说，你可以用它把数据导入电子表格），但它的显示格式却不怎么样，不便于直观检查和打印输出。下面就是那个 process_result_set()函数生成的一份输出报告：

```
Adams   John    Braintree   MA
Adams   John Quincy Braintree   MA
Arthur  Chester A.  Fairfield   VT
Buchanan    James   Mercersburg PA
Bush    George H.W. Milton  MA
Bush    George W.   New Haven   CT
Carter  James E.    Plains  GA
...
```

下面，我们将改写 process_result_set()函数，使它能够生成一份表格式的输出报告，把各数据列分别放在一个框里并加上标题。这个函数将以易于解释的格式显示同样的结果：

```
+------------+---------------+---------------------+-------+
| last_name  | first_name    | city                | state |
+------------+---------------+---------------------+-------+
| Adams      | John          | Braintree           | MA    |
| Adams      | John Quincy   | Braintree           | MA    |
| Arthur     | Chester A.    | Fairfield           | VT    |
| Buchanan   | James         | Mercersburg         | PA    |
| Bush       | George H.W.   | Milton              | MA    |
| Bush       | George W.     | New Haven           | CT    |
| Carter     | James E.      | Plains              | GA    |
...
+------------+---------------+---------------------+-------+
```

输出部分将按顺序做以下几件事。

(1) 确定各数据列的显示宽度。

(2) 打印标题行，用垂直线分隔，整个标题行放在上、下两行短划线之间。

(3) 把结果集每个数据行里的值依次打印出来，数据列之间用垂直线分隔并沿纵向对齐。如果是数字，则按居右方式打印；如果是 NULL 值，则打印为单词 NULL。

(4) 最后，把数据行总数打印在表格的下面。

这个练习很好地演示了结果集元数据的用法，除数据行中的数据值外，它还需要关于结果集本身的很多信息。

你也许会想："这好像与 mysql 程序生成输出报告的方式差不多嘛。"是的，的确如此。我希望大家把 mysql 程序的源代码与 process_result_set()函数的最终代码做一个对比。它们并不相同，但这种对比肯定会给你带来很多启发。

首先，为各数据列确定一个显示宽度。这项工作将由如下所示的代码负责完成。请注意：这段代码里的各种计算全部是根据结果集元数据进行的，没有涉及任何来自 MySQL 数据库的数据。

```
MYSQL_FIELD   *field;
unsigned long col_len;
unsigned int  i;

/* determine column display widths -- requires result set to be */
/* generated with mysql_store_result(), not mysql_use_result() */
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  field = mysql_fetch_field (res_set);
  col_len = strlen (field->name);
  if (col_len < field->max_length)
    col_len = field->max_length;
  if (col_len < 4 && !IS_NOT_NULL (field->flags))
    col_len = 4;  /* 4 = length of the word "NULL" */
  field->max_length = col_len;  /* reset column info */
}
```

为了计算出各数据列的显示宽度，这段代码将遍历与结果集里的各个数据列相对应的 MYSQL_FIELD 结构。在进入循环之前，先通过一个 mysql_field_seek()调用把自己定位到第一个结构处；在进入循环之后，再通过 mysql_fetch_field()调用逐个返回指向与当前数据列相对应的那个结构的指针。数据列的显示宽度是下面这 3 个值中的最大值，这 3 个值全部是根据结构里的元数据而获得的。

- field->name 的长度，即数据列标题。
- field->max_length，数据列中最长的那个数据值的长度。
- 字符串“NULL”的长度（如果数据列允许包含 NULL 值的话）。我们将根据 field->flags 来判断数据列是否允许包含 NULL 值。

请注意，在把数据列的显示宽度确定下来之后，我们把它赋值给了 MYSQL_FIELD 结构中的 max_length 成员。可是，MYSQL_FIELD 结构是我们从客户端库获得的，它是否允许修改呢？换个问法，MYSQL_FIELD 结构的内容是不是只读的呢？我得承认，我认为它是只读的。但 MySQL 发行版的有些客户程序也像这样对 max_length 进行过修改，所以我认为这样做也算合法。（如果你不想修改 max_length 成员，可以分配一个 unsigned long 数组来存放我们计算出来的显示宽度。）

在计算显示宽度时，有一个细节很关键。我们前面讲过，如果结果集是用 mysql_use_result()函数创建的，max_length 将没有任何实际意义。既然只有知道 max_length 的值才能确定数据列值的显示宽度，要想保证算法的正确操作，就应该用 mysql_store_result()函数来创建结果集。在使用 mysql_use_result()函数而非 mysql_store_result()函数的程序里，我们可以利用 MYSQL_FIELD 结构的 length 成员来达到目的，它可以告诉我们数据列值的最大长度是多少。

在知道了数据列的长度之后，就可以输出数据了。标题很容易处理。对于一个给定的数据列，只要简单地输出相应的 field 变量所指向的那个数据列信息结构的 name 成员就行了，别忘了用上刚才计算出来的宽度：

```
printf (" %-*s |", (int) field->max_length, field->name);
```

至于来自 MySQL 数据库的数据，我们将循环遍历结果集里的数据行，在每次循环中把当前数据行的数据列值依次打印出来。数据列值的打印工作也有一个地方需要注意，因为它可能是 NULL 值，也可能是数值（数值必须按居右格式打印）。下面是我们用来打印数据列值的代码，其中 row[i]存放

着数据值，指针 field 则指向数据列信息：

```
if (row[i] == NULL)       /* print the word "NULL" */
  printf (" %-*s |", (int) field->max_length, "NULL");
else if (IS_NUM (field->type))  /* print value right-justified */
  printf (" %*s |", (int) field->max_length, row[i]);
else              /* print value left-justified */
  printf (" %-*s |", (int) field->max_length, row[i]);
```

如果某个数据列的 field->type 表明它是数值类型（INT、FLOAT、DECIMAL 等），IS_NUM()宏的求值结果将为真。

用来打印结果集数据的最终代码如下所示。因为要多次打印短划线，所以我们还编写了一个 print_dashes()函数，这要比在多个地方重复使用同样的代码来打印短划线省力。

```
void
print_dashes (MYSQL_RES *res_set)
{
MYSQL_FIELD   *field;
unsigned int  i, j;

  mysql_field_seek (res_set, 0);
  fputc ('+', stdout);
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    field = mysql_fetch_field (res_set);
    for (j = 0; j < field->max_length + 2; j++)
      fputc ('-', stdout);
    fputc ('+', stdout);
  }
  fputc ('\n', stdout);
}

void
process_result_set (MYSQL *conn, MYSQL_RES *res_set)
{
MYSQL_ROW     row;
MYSQL_FIELD   *field;
unsigned long col_len;
unsigned int  i;

  /* determine column display widths -- requires result set to be */
  /* generated with mysql_store_result(), not mysql_use_result() */
  mysql_field_seek (res_set, 0);
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    field = mysql_fetch_field (res_set);
    col_len = strlen (field->name);
    if (col_len < field->max_length)
      col_len = field->max_length;
    if (col_len < 4 && !IS_NOT_NULL (field->flags))
      col_len = 4;  /* 4 = length of the word "NULL" */
    field->max_length = col_len;  /* reset column info */
  }
```

7

```
    print_dashes (res_set);
    fputc ('|', stdout);
    mysql_field_seek (res_set, 0);
    for (i = 0; i < mysql_num_fields (res_set); i++)
    {
      field = mysql_fetch_field (res_set);
      printf (" %-*s |", (int) field->max_length, field->name);
    }
    fputc ('\n', stdout);
    print_dashes (res_set);

    while ((row = mysql_fetch_row (res_set)) != NULL)
    {
      mysql_field_seek (res_set, 0);
      fputc ('|', stdout);
      for (i = 0; i < mysql_num_fields (res_set); i++)
      {
        field = mysql_fetch_field (res_set);
        if (row[i] == NULL)       /* print the word "NULL" */
          printf (" %-*s |", (int) field->max_length, "NULL");
        else if (IS_NUM (field->type))  /* print value right-justified */
          printf (" %*s |", (int) field->max_length, row[i]);
        else              /* print value left-justified */
          printf (" %-*s |", (int) field->max_length, row[i]);
      }
      fputc ('\n', stdout);
    }
    print_dashes (res_set);
    printf ("Number of rows returned: %lu\n",
            (unsigned long) mysql_num_rows (res_set));
  }
```

MySQL 客户端库还提供了其他一些方法供你访问数据列信息结构。比如说，上面的代码曾使用下面这样的循环语句来多次访问这些结构：

```
mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  field = mysql_fetch_field (res_set);
  ...
}
```

mysql_field_seek()加mysql_fetch_field()的组合并非访问MYSQL_FIELD结构的唯一手段。在线资源附录 G 介绍了 mysql_fetch_fields()和 mysql_fetch_field_direct()等用来访问数据列信息结构的其他方法。

使用 metadata 程序来输出某给定结果集的元数据

sampdb 发行版本包含一个名为 metadata 的程序的源代码，你可以编译并运行它以查看各种语句所产生的元数据。它会提示用户输入一条 SQL 语句并执行，但它输出的是结果集的元数据而不是结果集的内容。

7.4.7 对特殊字符和二进制数据进行编码

当程序假设用户输入的语句合法或者程序可以向用户报告出错消息时，程序会执行用户输入的语句。比如说，如果某个用户想在一个用引号括起来的字符串里使用引号字符本身，必须双写那个引号字符或在引号字符前加上一个反斜线字符，如下所示：

```
'O''Malley'
'O\'Malley'
```

自行构造语句的应用程序必须注意这个问题。本节将讨论如何处理字符串值里的引号问题和二进制数据。

1. 如何处理包含特殊字符的字符串

如果不加处理地把包含引号、NULL 字节、反斜线的数据值放到一条语句里，在试图执行这条语句时就会遇到麻烦。下面将对这一问题的本质和解决办法进行讨论。

假设在你构造的SELECT查询里用到了一个名为name_val的变量，而这个变量的值是一个以NULL字节结尾的字符串，如下所示：

```
char stmt_buf[1024];

sprintf (stmt_buf, "SELECT * FROM mytbl WHERE name='%s'", name_val);
```

那么，万一变量 name_val 的值是 O'Malley, Brain 之类的东西，你构造出来的语句就是非法的，因为用引号括起来的字符串里又出现了一个引号字符：

```
SELECT * FROM mytbl WHERE name='O'Malley, Brian'
```

你必须把字符串内部的引号当做特殊情况来处理，否则，服务器就会把它解释为字符串的结束标志。SQL 语言标准给出的解决办法是双写字符串里的引号字符。MySQL 既支持这一做法，也允许你使用一个反斜线字符对引号字符进行转义，所以你可以用下面两种格式之一来写出这条语句：

```
SELECT * FROM mytbl WHERE name='O''Malley, Brian'
SELECT * FROM mytbl WHERE name='O\'Malley, Brian'
```

这类问题可以用 mysql_real_escape_string()函数来解决，这个函数将对特殊字符进行编码，使它们适合用在以引号括起来的字符串里。mysql_real_escape_string()函数将把 NULL 字节、单引号（'）、双引号（"）、反斜线（\）、换行符、回车符、Ctrl-Z 等字符视为特殊字符。（Ctrl-Z 是 Windows 里的特殊字符，它通常被用做文件结束标志。）

那么，哪些场合应该使用 mysql_real_escape_string()函数呢？最保险的答案是“任何场合”。不过，如果你对数据的格式和内容都很有把握（比如，你事先做过一些检查），就不必对它们进行编码。例如，如果你正在处理的字符串代表的是一些合法的电话号码——即仅由数字和短线组成，你当然没必要去调用这个函数。至于其他情况，你还是慎重些好。

mysql_real_escape_string()函数将把特殊字符编码为一个由两个字符构成的序列，第一个字符是反斜线（\）。比如说，NULL 字节将被编码为\0，其中的 0 是可打印的 ASCII 字符 0 而不再是 NULL 字节。反斜线、单引号、双引号将分别被编码为“\\”、“\'”和“\"”。

mysql_real_escape_string()函数的调用语法如下所示：

```
to_len = mysql_real_escape_string (conn, to_str, from_str, from_len);
```

mysql_real_escape_string()对 from_str 进行编码并把结果写到 to_str 里。它还会在

to_str 的末尾加上一个 NULL 字节，这为我们使用 strcpy()、strlen()、printf()等函数来进一步处理那个结果字符串提供了方便。

from_str 指向一个 char 缓冲区，其内容是将被编码的字符串；这个字符串允许包含包括二进制数据在内的任何东西。to_str 指向一个已经存在的 char 缓冲区，其内容将是编码后的结果字符串。注意，千万不要传递未经初始化的指针或者 NULL 指针，mysql_real_escape_string()函数是不会替你去分配内存的！to_str 指向的缓冲区的长度必须至少有(form_len * 2)+1 个字节长，可能 from_str 里的每个字符都需要被编码为一个由两个字符构成的序列，多出来的最后一个字节是为字符串结束标志（NULL 字节）而保留的。

from_len 和 to_len 都是 unsigned long 类型的值。from_len 给出的是 from_str 缓冲区里的数据长度，这个长度值是必不可少的，因为 from_str 允许包含 NULL 字节，所以你不能把它当做以 NULL 字节结尾的字符串。to_len，也就是 mysql_real_escape_string()函数的返回值，是编码结果字符串的实际长度，作为其结束标志的那个 NULL 字节不计算在内。

当 mysql_real_escape_string()调用返回时，from_str 里的 NULL 字节将全都被编码为可打印的\0 序列，所以我们可以把 to_str 里的编码结果当做以 NULL 字节结尾的字符串。

现在改写一下用来生成 SELECT 查询语句的代码，让它在遇到包含引号的值时也能够正确工作。我们可以这样做：

```
char stmt_buf[1024], *p;

p = strcpy (stmt_buf, "SELECT * FROM mytbl WHERE name='");
p += strlen (p);
p += mysql_real_escape_string (conn, p, name_val, strlen (name_val));
*p++ = '\'';
*p = '\0';
```

我得承认，这段代码不太容易看明白。下面这段代码就简洁多了，但这是有代价的——得多使用一个缓冲区：

```
char stmt_buf[1024], buf[1024];

(void) mysql_real_escape_string (conn, buf, name_val, strlen (name_val));
sprintf (stmt_buf, "SELECT * FROM mytbl WHERE name='%s'", buf);
```

有一点需要大家特别注意：传递给 mysql_real_escape_string()函数的缓冲区必须是真正存在的。下面这个例子违背了这一原则：

```
char *from_str = "some string";
char *to_str;
unsigned long len;

len = mysql_real_escape_string (conn, to_str, from_str, strlen (from_str));
```

看出问题了吗？to_str 本该指向一个已经存在的缓冲区，但它却没有，它没有被初始化，而这意味着它将随机指向一个不确定的位置。再强调一次，如果不想把内存弄成一团糟，就千万不要把一个未经初始化的指针传递给 mysql_real_escape_string()函数的 to_str 参数。

2. 如何处理二进制数据

在语句里使用二进制数据（例如一个用来把图像存入数据库的应用程序）也存在着类似的问题。

因为二进制值允许包含任何字符（包括 NULL 字节、引号或反斜线），所以把它们直接放到语句里并不安全。

处理二进制数据离不开 mysql_real_escape_string()函数。本节将演示如何使用从文件读出的图像数据来处理二进制数据。这里的讨论同样适用于任何其他形式的二进制数据。

假设你需要从文件里读出图像并存入一个名为 picture 的数据表，其中每幅图像都有一个独一无二的标识符。MEDIUMBLOB 类型对长度小于 16 MB 的二进制值来说是个很好的选择，所以你可以定义一个如下所示的数据表：

```
CREATE TABLE picture
(
  pict_id   INT NOT NULL PRIMARY KEY,
  pict_data MEDIUMBLOB
);
```

如果标识号和指针已赋给了包含图像数据的打开的文件，把从文件里读出的图像加载到 picture 数据表里的工作将由 load_image()函数完成。下面就是它的代码：

```
int
load_image (MYSQL *conn, int id, FILE *f)
{
char          stmt_buf[1024*1024], buf[1024*10], *p;
unsigned long from_len;
int           status;

  /* begin creating an INSERT statement, adding the id value */
  sprintf (stmt_buf,
           "INSERT INTO picture (pict_id,pict_data) VALUES (%d,'",
           id);
  p = stmt_buf + strlen (stmt_buf);
  /* read data from file in chunks, encode each */
  /* chunk, and add to end of statement */
  while ((from_len = fread (buf, 1, sizeof (buf), f)) > 0)
  {
    /* don't overrun end of statement buffer! */
    if (p + (2*from_len) + 3 > stmt_buf + sizeof (stmt_buf))
    {
      print_error (NULL, "image is too big");
      return (1);
    }
    p += mysql_real_escape_string (conn, p, buf, from_len);
  }
  *p++ = '\'';
  *p++ = ')';
  status = mysql_real_query (conn, stmt_buf, (unsigned long) (p - stmt_buf));
  return (status);
}
```

load_image()函数分配的语句缓冲区不算大（1 MB），所以它只能用来加载比较小的图像。在实际工作中，你可以根据图像文件的大小动态地为这个缓冲区分配内存。

从数据库取回一幅图像数据（或任意二进制值）没有当初把它存入数据库时那么麻烦。原始格式的数据值就在 MYSQL_ROW 变量里，其长度可以通过调用 mysql_fetch_lengths()函数获得。只要注

意把这类数据值当做一个计数字符串来处理就不会有什么大问题，千万不要把它当做一个以空字节结束的字符串来对待。

7.5　交互式语句执行程序

在这一小节里，我们将把此前开发的代码结合起来去编写一个简单的交互式语句执行客户程序 exec_stmt。这个程序将接收你输入的语句，并用通用查询处理程序 process_statement()执行，再用上一节里开发的格式化程序 process_result_set()把查询结果打印出来。

exec_stmt 程序与 mysql 客户程序很相似，当然在功能上没有那么丰富。exec_stmt 程序对自己的输入有以下几项限制。

- 每个输入行只能包含一条完整的语句。
- 语句不需要以分号或者\g 结尾。
- 只支持一条 SQL 命令，即用来结束程序运行的 quit 和\q。还可以使用 Ctrl-D 组合键来退出运行。

有了前面那些成果，exec_stmt 程序就很容易写了（新代码大概只有十几行）。客户程序框架（client2.c）和我们已经完成的其他函数几乎提供了所有必需的东西。我们只需再增加一个用来接收输入行并执行它们的循环就大功告成了。

编写 exec_stmt 程序的第一步是把客户程序框架 client2.c 复制到 exec_stmt.c。然后再把 process_statement()、process_result_set()和 print_dashes()函数添加到 exec_stmt.c 里。最后，在 exec_stmt.c 里，在 main()函数中找到如下所示的那一行：

```
/* ... issue statements and process results here ... */
```

把这一行替换为如下所示的 while 循环：

```
while (1)
{
  char  buf[10000];

  fprintf (stderr, "query> ");                    /* print prompt */
  if (fgets (buf, sizeof (buf), stdin) == NULL) /* read statement */
    break;
  if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
    break;
  process_statement (conn, buf);                 /* execute it */
}
```

现在，把 exec_stmt.c 编译为 exec_stmt.o，再把 exec_stmt.o 与客户端库链接起来生成 exec_stmt，我们就将得到一个能够执行任何语句并显示其结果的交互式 MySQL 客户程序。下面几个例子演示了 exec_stmt 程序的工作情况，包括 SELECT 查询和非 SELECT 查询，还故意给出了几个有错误的语句：

```
% ./exec_stmt
query> USE sampdb
Number of rows affected: 0
query> SELECT DATABASE(), USER()
+------------+-------------------+
| DATABASE() | USER()            |
+------------+-------------------+
```

```
| sampdb      | sampadm@localhost |
+------------+-------------------+
Number of rows returned: 1
query> SELECT COUNT(*) FROM president
+----------+
| COUNT(*) |
+----------+
|       42 |
+----------+
Number of rows returned: 1
query> SELECT last_name, first_name FROM president ORDER BY last_name LIMIT 3
+-----------+-------------+
| last_name | first_name  |
+-----------+-------------+
| Adams     | John        |
| Adams     | John Quincy |
| Arthur    | Chester A.  |
+-----------+-------------+
Number of rows returned: 3
query> CREATE TABLE t (i INT)
Number of rows affected: 0
query> SELECT j FROM t
Could not execute statement
Error 1054 (42S22): Unknown column 'j' in 'field list'
query> USE mysql
Could not execute statement
Error 1044 (42000): Access denied for user 'sampadm'@'localhost' to
database 'mysql'
```


7.6　怎样编写具备 SSL 支持的客户程序

MySQL 包含 SSL 支持功能，你可以在自己的程序里利用这些功能通过安全连接去访问服务器。为展示如何完成这一功能，本节将修改 exec_stmt 以生成一个类似的客户程序 exec_stmt_ssl。这两个程序看上去极其相似，但后者多了一项建立加密连接的能力。要使 exec_stmt_ssl 正常工作，MySQL 必须有 SSL 支持组件，并且服务器在启动时已经通过适当的选项把自己的证书和密钥文件设定好了。此外，你还需要把客户端的证书和密钥文件准备好。详情请参见 13.3 节。

这个程序的源代码文件 exec_stmt_ssl.c 已经收录在 sampdb 发行版本里了，你可以直接用它去建立 exec_stmt_ssl 客户程序。从 exec_stmt.c 文件开始去创建 exec_stmt_ssl.c 文件的过程如下所示。

(1) 将 exec_stmt.c 复制到 exec_stmt_ssl.c 文件。以下步骤都将在 exec_stmt_ssl.c 文件里进行。

(2) 为了让编译器检测出 SSL 支持是否可用，MySQL 在头文件 my_config.h 里相应地定义了一个名为 HAVE_OPENSSL 的符号。在编写与 SSL 有关的代码时，你应该使用如下所示的构造。这样，如果无法使用 SSL，代码将被忽略。

```
#ifdef HAVE_OPENSSL
  ...SSL-related code here...
#endif
```

因为 my_global.h 包含 my_config.h，而 exec_stmt_ssl.c 文件已经包含 my_global.h，所以你就不必再明确地包含 my_config.h 文件了。

(3) 修改以选项信息结构为元素的 my_opts 数组，把与 SSL 有关的标准选项（--ssl-ca、--ssl-key 等）也添加进去。最简便的办法是用一条#include 指令把 sslopt-longopts.h 头文件的内容包括到 my_opt 数组里来。修改后的 my_opts 数组如下所示：

```
static struct my_option my_opts[] =   /* option information structures */
{
  {"help", '?', "Display this help and exit",
  NULL, NULL, NULL,
  GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0},
  {"host", 'h', "Host to connect to",
  (uchar **) &opt_host_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"password", 'p', "Password",
  (uchar **) &opt_password, NULL, NULL,
  GET_STR, OPT_ARG, 0, 0, 0, 0, 0, 0},
  {"port", 'P', "Port number",
  (uchar **) &opt_port_num, NULL, NULL,
  GET_UINT, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"socket", 'S', "Socket path",
  (uchar **) &opt_socket_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"user", 'u', "User name",
  (uchar **) &opt_user_name, NULL, NULL,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},

#include <sslopt-longopts.h>

  { NULL, 0, NULL, NULL, NULL, NULL, GET_NO_ARG, NO_ARG, 0, 0, 0, 0, 0, 0 }
};
```

sslopt-longopts.h 是一个 MySQL 公共头文件，它的内容如下所示（格式稍有调整）：

```
#ifdef HAVE_OPENSSL
  {"ssl", OPT_SSL_SSL,
  "Enable SSL for connection (automatically enabled with other flags).
  Disable with --skip-ssl.",
  (uchar **) &opt_use_ssl, (uchar **) &opt_use_ssl, 0,
  GET_BOOL, NO_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-ca", OPT_SSL_CA,
  "CA file in PEM format (check OpenSSL docs, implies --ssl).",
  (uchar **) &opt_ssl_ca, (uchar **) &opt_ssl_ca, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-capath", OPT_SSL_CAPATH,
  "CA directory (check OpenSSL docs, implies --ssl).",
  (uchar **) &opt_ssl_capath, (uchar **) &opt_ssl_capath, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-cert", OPT_SSL_CERT, "X509 cert in PEM format (implies --ssl).",
  (uchar **) &opt_ssl_cert, (uchar **) &opt_ssl_cert, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-cipher", OPT_SSL_CIPHER, "SSL cipher to use (implies --ssl).",
  (uchar **) &opt_ssl_cipher, (uchar **) &opt_ssl_cipher, 0,
  GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
  {"ssl-key", OPT_SSL_KEY, "X509 key in PEM format (implies --ssl).",
  (uchar **) &opt_ssl_key, (uchar **) &opt_ssl_key, 0,
```

```
    GET_STR, REQUIRED_ARG, 0, 0, 0, 0, 0, 0},
#ifdef MYSQL_CLIENT
  {"ssl-verify-server-cert", OPT_SSL_VERIFY_SERVER_CERT,
   "Verify server's \"Common Name\" in its cert against hostname used
   when connecting. This option is disabled by default.",
   (uchar **) &opt_ssl_verify_server_cert,
   (uchar **) &opt_ssl_verify_server_cert, 0,
   GET_BOOL, NO_ARG, 0, 0, 0, 0, 0, 0},
#endif
#endif /* HAVE_OPENSSL */
```

(4) 在上一步里，sslopt-longopts.h 在定义选项结构时使用了 OPT_SSL_SSL、OPT_SSL_KEY 等值。这些值的用途是充当短选项代码，它们必须由你的程序来定义，具体而言，你只要把下面这些代码添加到 my_opts 数组定义的前面就行了：

```
#ifdef HAVE_OPENSSL
enum options_client
{
  OPT_SSL_SSL=256,
  OPT_SSL_KEY,
  OPT_SSL_CERT,
  OPT_SSL_CA,
  OPT_SSL_CAPATH,
  OPT_SSL_CIPHER,
  OPT_SSL_VERIFY_SERVER_CERT
};
#endif
```

在编写你自己的程序时，如果某给定程序也为其他一些选项定义了代码，请务必保证这些 OPT_SSL_XXX 形式的符号与其他代码有着不同的值。

(5) sslopt-longopts.h 定义了一些与 SSL 有关的选项结构，你得把这些选项的值保存在一些变量里。为了声明这些变量，需要用一条#include 指令把 sslopt-vars.h 包括到你的程序里 my_opts 数组定义的前面。sslopt-vars.h 的内容如下所示：

```
#ifdef HAVE_OPENSSL
static my_bool opt_use_ssl  = 0;
static char *opt_ssl_ca     = 0;
static char *opt_ssl_capath = 0;
static char *opt_ssl_cert   = 0;
static char *opt_ssl_cipher = 0;
static char *opt_ssl_key    = 0;
#ifdef MYSQL_CLIENT
static my_bool opt_ssl_verify_server_cert= 0;
#endif
#endif
```

(6) 在 get_one_option()例程的末尾附近增加一行代码把 sslopt-case.h 包括进来，如下所示：

```
static my_bool
get_one_option (int optid, const struct my_option *opt, char *argument)
{
  switch (optid)
  {
  case '?':
```

```
        my_print_help (my_opts);  /* print help message */
        exit (0);
      case 'p':                   /* password */
        if (!argument)            /* no value given; solicit it later */
          ask_password = 1;
        else                      /* copy password, overwrite original */
        {
          opt_password = strdup (argument);
          if (opt_password == NULL)
          {
            print_error (NULL, "could not allocate password buffer");
            exit (1);
          }
          while (*argument)
            *argument++ = 'x';
          ask_password = 0;
        }
        break;
    #include <sslopt-case.h>
      }
      return (0);
    }
```

sslopt-case.h 的内容是一些用在 switch()语句里的 case 子句。这些子句负责检测是否有 SSL 选项，如果有，就设置 opt_use_ssl 变量。这个文件的内容如下所示：

```
#ifdef HAVE_OPENSSL
  case OPT_SSL_KEY:
  case OPT_SSL_CERT:
  case OPT_SSL_CA:
  case OPT_SSL_CAPATH:
  case OPT_SSL_CIPHER:
  /*
    Enable use of SSL if we are using any ssl option
    One can disable SSL later by using --skip-ssl or --ssl=0
  */
    opt_use_ssl= 1;
    break;
#endif
```

这样做的效果是：在选项处理工作结束后，程序可以通过检测 opt_use_ssl 值来判断用户是否想使用安全连接。

在完成以上步骤之后，常见的 load_defaults()和 handle_options()函数就能识别出 SSL 相关的选项并自动设置好它们的值了。现在，还剩下最后一件事情：如果选项表明用户想建立 SSL 连接，那么应该在连接服务器之前传递 SSL 选项信息。这个工作可以调用 mysql_ssl_set()函数来完成，这个函数必须安排在调用 mysql_init()函数之后、调用 mysql_real_connect()函数之前。如下所示：

```
  /* initialize connection handler */
  conn = mysql_init (NULL);
  if (conn == NULL)
  {
    print_error (NULL, "mysql_init() failed (probably out of memory)");
```

```
    exit (1);
  }

#ifdef HAVE_OPENSSL
  /* pass SSL information to client library */
  if (opt_use_ssl)
    mysql_ssl_set (conn, opt_ssl_key, opt_ssl_cert, opt_ssl_ca,
                   opt_ssl_capath, opt_ssl_cipher);
#if (MYSQL_VERSION_ID >= 50023 && MYSQL_VERSION_ID < 50100) \
    || MYSQL_VERSION_ID >= 50111
  mysql_options (conn,MYSQL_OPT_SSL_VERIFY_SERVER_CERT,
                 (char*)&opt_ssl_verify_server_cert);
#endif
#endif

  /* connect to server */
  if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
      opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
  {
    print_error (conn, "mysql_real_connect() failed");
    mysql_close (conn);
    exit (1);
  }
```

代码没有测试 mysql_ssl_set()调用是否返回了一个错误。如果你传递给这个函数的信息有问题，你就会在发出 mysql_real_connect()调用之后看到一条出错信息。MYSQL_OPT_SSL_VERIFY_SERVER_CERT 是从 MySQL 5.0 系列的 5.0.23 版和 MySQL 5.1 系列的 5.1.11 版开始的新增功能，上段代码在判断是否要调用 mysql_options()函数时进行的复杂测试是为了保证有关代码在不同版本的 MySQL 系统上都可以正常运行。

现在，编译 exec_stmt_ssl.c 文件以生成 exec_stmt_ssl 程序，然后运行它。如果 mysql_real_connect()调用成功，你就可以发出语句了。如果你在调用 exec_stmt_ssl 时使用了适当的 SSL 选项，你与服务器之间的通信就将通过一条加密连接进行。如果你想知道情况是否如此，可以发出一个下面这样的语句：

```
SHOW STATUS LIKE 'Ssl_cipher'
```

如果 Ssl_cipher 变量的值非空，就说明你正使用着某种加密算法。（为了让大家省点事，sampdb 发行版本里的 exec_stmt_ssl 会替你发出这个语句并报告其结果。）

7.7 嵌入式服务器库的使用

完整的 MySQL 软件还包括一个名为 libmysqld 的嵌入式服务器库，这个库所提供的功能可以让我们把 MySQL 服务器链接（或者说嵌入）到各种应用程序里。这使我们可以开发出自带 MySQL 服务器端功能的独立应用程序，而不是那些只能作为一个客户通过网络另行连接某个服务器程序的普通 MySQL 应用程序。

如果你想编写一个嵌入了服务器的应用程序，就必须满足两个条件。首先，必须安装嵌入式 MySQL 服务器库：

- ❑ 如果从源代码开始构建，请在运行 configure 时使用--with-embeded-server 选项，这同样

适用于 MySQL 5.0 和 5.1；

- 如果使用的是二进制发行版本，请使用 MySQL 5.1，因为 5.0 版不包含 libmysqld，当前的 5.1 版则包含 libmysqld。如果使用的是 RPM 包，请务必安装单独的一个嵌入式 RPM。

其次，你还需要在应用程序里增加一些用来启动和关闭服务器的代码。

在确认这两项要求都已得到满足之后，只需编译该应用程序并链接嵌入式服务器库（-lmysqld）即可，千万不要像往常那样链接 MySQL 客户库（-lmysqlclient）。MySQL 服务器库的设计十分巧妙，如果你编写了一个应用程序来使用它，只要与相应的库分别链接，就可以得到一个嵌入版或是一个客户/服务器版的应用程序。之所以如此，是因为 MySQL 客户库提供了必要的接口函数，它们可以按客户/服务器通信方式完成初始化和终止处理工作，而不是按照与一个嵌入式服务器进行通信的要求完成这些工作。

7.7.1 编写内建了服务器的应用程序

编写一个内建有服务器的应用程序与编写一个将运行在客户/服务器环境里的应用程序并没有太大的区别。事实上，即使你当初打算编写的是一个将运行在客户/服务器环境里的客户程序，你也可以轻易地将它转换为使用嵌入式服务器的程序。下面，我们就介绍如何生成嵌入式应用程序 embapp，首先从 exec_stmt.c 开始。

(1) 把 exec_stmt.c 文件复制为 embapp.c 文件。接下来的步骤在 embapp.c 文件里完成。（之所以从 exec_stmt.c 文件而不是 exec_stmt_ssl.c 文件开始，是因为所有通信将只发生在同一个应用程序的内部，没有必要使用 SSL。）

(2) 把 mysql_embed.h 文件添加到你的应用程序所使用的 MySQL 头文件当中：

```
#include <my_global.h>
#include <my_sys.h>
#include <m_string.h>   /* for strdup() */
#include <mysql.h>
#include <mysql_embed.h>
#include <my_getopt.h>
```

(3) 一个嵌入式应用程序同时包含着一个客户端和一个服务器端，所以它既可以处理供客户端使用的选项，也可以处理供服务器端使用的选项。比如说，假设 embapp 程序需要为它的客户部分去读取选项文件里的[client]和[embapp]选项组，你就需要把 client_groups 数组的定义修改为如下所示的样子：

```
static const char *client_groups[] =
{
  "client", "embapp", NULL
};
```

这些选项组里的选项将由 load_defaults()和 handle_options()函数按常见方式处理。接下来，我们还要定义一组供服务器端使用的选项组。根据规定，这组选项组将包括[server]、[embedded]和[appname_SERVER]，其中，appname 表示应用程序的名字。具体到 embapp 程序，供它专用的选项组就是[embapp_SERVER]，这组选项组的名字定义为如下所示：

```
static const char *server_groups[] =
{
  "server", "embedded", "embapp_server", NULL
```

```
};
```

(4) 在初始化与服务器的通信之前，调用 mysql_init() 函数，修改这个调用让它把需要服务器处理的选项全部传递过去。必要的准备工作应该在调用 mysql_init() 函数之前安排妥当。

(5) 在结束与服务器的通信之后，调用 mysql_library_end()，最好是在调用 mysql_close() 函数之后。

完成上述修改后，embapp.c 里的 main() 函数将是如下所示的样子：

```
int
main (int argc, char *argv[])
{
int opt_err;

  MY_INIT (argv[0]);
  load_defaults ("my", client_groups, &argc, &argv);

  if ((opt_err = handle_options (&argc, &argv, my_opts, get_one_option)))
    exit (opt_err);

  /* solicit password if necessary */
  if (ask_password)
    opt_password = get_tty_password (NULL);

  /* get database name if present on command line */
  if (argc > 0)
  {
    opt_db_name = argv[0];
    --argc; ++argv;
  }

  /* initialize embedded server library */
  if (mysql_library_init (0, NULL, (char **) server_groups))
  {
    print_error (NULL, "mysql_library_init() failed");
    exit (1);
  }

  /* initialize connection handler */
  conn = mysql_init (NULL);
  if (conn == NULL)
  {
    print_error (NULL, "mysql_init() failed (probably out of memory)");
    exit (1);
  }

  /* connect to server */
  if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
      opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
  {
    print_error (conn, "mysql_real_connect() failed");
    mysql_close (conn);
    exit (1);
  }
```

```
  while (1)
  {
    char  buf[10000];

    fprintf (stderr, "query> ");                  /* print prompt */
    if (fgets (buf, sizeof (buf), stdin) == NULL) /* read statement */
      break;
    if (strcmp (buf, "quit\n") == 0 || strcmp (buf, "\\q\n") == 0)
      break;
    process_statement (conn, buf);                /* execute it */
  }

  /* disconnect from server */
  mysql_close (conn);
  /* shut down embedded server library */
  mysql_library_end ();
  exit (0);
}
```

7.7.2　生成应用程序可执行二进制文件

如果要生成一个内建有服务器的 embapp 可执行二进制文件，就需要用-lmysqld 库而不是-lmysqlclient 库来链接。这里是 mysql_config 工具大显身手的地方。它能告诉你需要使用哪些标志来链接普通客户端库，也能告诉你需要使用哪些标志来链接嵌入了服务器的库：

```
% mysql_config --libmysqld-libs
 -L'/usr/local/mysql/lib/mysql' -lmysqld -lz -lm
```

用来生成嵌入版 embapp 程序的命令将是：

```
% gcc -c `mysql_config --include` embapp.c
% gcc -o embapp embapp.o `mysql_config --libmysqld-libs`
```

说明　*在执行这些命令的时候，你或许会发现有必要使用诸如 g++之类的 C++编译器来代替 C 编译器。*

到了这一步，你就得到了一个嵌入式应用程序，它有你需要用来访问 MySQL 数据库的一切东西。不过，当你运行 embapp 程序时，千万不要让它与正在同一台机器里运行的其他服务器使用相同的数据目录。

同样，在 Unix 系统上，必须通过一个有权访问数据目录的账户去运行这个应用程序。如果你是以数据目录的属主身份登录的，想运行 embapp 程序当然不会有问题。也可以把 embapp 程序转换为一个设置用户标识符的程序，让它在启动时把自己的用户 ID 自动切换为指定用户。比如说，如果想让 embapp 程序在运行时具备用户 mysql 的权限，请以根用户 root 的身份执行以下命令：

```
# chown mysql embapp
# chmod 4755 embapp
```

如果想生成一个将在客户/服务器环境里运行的非嵌入的程序，就把它与普通客户端库链接起来。如下所示：

```
% gcc -c `mysql_config --include` embapp.c
```

```
% gcc -o embapp embapp.o `mysql_config --libs`
```

MySQL 客户端库里的 mysql_library_init()和 mysql_library_end()函数可以按客户/服务器通信方式完成初始化和终处理工作，而不是按照与一个嵌入式服务器进行通信的要求完成这些工作。

7.8 一次执行多条语句

MySQL 客户端库还支持一次执行多条语句。这使我们可以把一个由多条语句构成的字符串——语句之间要用分号隔开——发送给服务器，然后依次检索和处理各个语句的结果集。

这种多语句执行能力并不是默认启用的，所以必须明确地告诉服务器你打算使用这项功能。有两个办法可以做到这一点。第一个办法是在连接服务器时把 CLIENT_MULTI_STATEMENTS 选项添加到 mysql_real_connect()函数的标志参数当中，如下所示：

```
opt_flags |= CLIENT_MULTI_STATEMENTS;
if (mysql_real_connect (conn, opt_host_name, opt_user_name, opt_password,
    opt_db_name, opt_port_num, opt_socket_name, opt_flags) == NULL)
{
  print_error (conn, "mysql_real_connect() failed");
  mysql_close (conn);
  exit (1);
}
```

另一个办法是用 mysql_set_server_option()函数为一个现有的连接启用这项能力，如下所示：

```
if (mysql_set_server_option (conn, MYSQL_OPTION_MULTI_STATEMENTS_ON) != 0)
  print_error (conn, "Could not enable multiple-statement execution");
```

哪一种办法更好呢？如果程序里没有使用存储过程，这两种办法都适用。如果程序里使用了存储过程并用 CALL 语句返回了结果集，第一种办法更好。这是因为 CLIENT_MULTI_STATEMENTS 选项会同时打开 CLIENT_MULTI_RESULTS 选项。如果后者没被启用，当某个存储过程试图返回一个结果集时就会导致程序出错。（总的来说，还是把 CLIENT_MULTI_STATEMENTS 选项添加到 mysql_real_connect()函数的标志参数当中去的办法更好，因为它能明白无误地表明你启用了这个选项。）

在处理多个结果集时，下面两个函数是你检查结果检索操作的当前状态的基本武器。

- mysql_more_results()函数。如果还有更多的结果，它将返回一个非零值；否则，返回 0。
- mysql_next_result()函数。如果还有更多的结果，它在返回一个状态值的同时还会对下一个结果集的检索工作进行初始化。如果还有更多的结果，状态值是 0；如果没有，返回-1；如果发生错误，返回一个大于零的值。

可以把结果检索代码放在一个循环里来使用这些函数。用往常的代码检索出一个结果之后，记得要检查一下是否还有需要检索的结果。如果有，那就再循环一次。如果没有，退出循环。根据具体编写的循环语句，或许根本用不着去调用 mysql_more_results()函数，因为 mysql_next_result()函数的返回值也可以让你知道是否还有更多的结果。

7.4.3 节曾经编写过一个 process_statement()函数，它可以用来执行一条语句并检索其结果，或是显示受其影响的数据行的个数。通过把结果检索代码放入一个循环和使用 mysql_next_result()函数，我们可以编写一个与之类似的 process_mutil_statement()函数来检索多个结果：

```
void
```

```
process_multi_statement (MYSQL *conn, char *stmt_str)
{
MYSQL_RES *res_set;
int       status;
int       keep_going = 1;

  if (mysql_query (conn, stmt_str) != 0)  /* the statement(s) failed */
  {
    print_error (conn, "Could not execute statement(s)");
    return;
  }

  /* the statement(s) succeeded; enter result-retrieval loop */
  do {
    /* determine whether current statement returned data */
    res_set = mysql_store_result (conn);
    if (res_set)      /* a result set was returned */
    {
      /* process rows and then free the result set */
      process_result_set (conn, res_set);
      mysql_free_result (res_set);
    }
    else              /* no result set was returned */
    {
      /*
       * does the lack of a result set mean that the statement didn't
       * return one, or that it should have but an error occurred?
       */
      if (mysql_field_count (conn) == 0)
      {
        /*
         * statement generated no result set (it was not a SELECT,
         * SHOW, DESCRIBE, etc.); just report rows-affected value.
         */
        printf ("Number of rows affected: %lu\n",
                (unsigned long) mysql_affected_rows (conn));
      }
      else  /* an error occurred */
      {
        print_error (conn, "Could not retrieve result set");
        keep_going = 0;
      }
    }
    /* determine whether more results exist */
    /* 0 = yes, -1 = no, >0 = error */
    status = mysql_next_result (conn);
    if (status != 0)    /* no more results, or an error occurred */
    {
      keep_going = 0;
      if (status > 0)   /* error */
        print_error (conn, "Could not execute statement");
    }
  } while (keep_going);
}
```

如果愿意，你可以只测试 mysql_next_result()函数的返回值是不是 0，如果不是则退出循环。这个简单的策略存在着这样一个不足：当它告诉你没有更多结果的时候，你无法知道是已经正常地到达了末尾还是因为发生了错误。换句话说，你无法判断是否应该显示一条出错消息。

7.9 使用服务器端预处理语句

在本章前面的内容里，用来处理 SQL 语句的代码以 MySQL 客户端库提供的函数为基础，而那些函数都是以字符串形式来发送和检索所有信息的。本节将讨论如何使用二进制客户/服务器协议。二进制协议支持服务器端预处理语句并让我们能够以数据值的原始格式来传输它们。

并非所有语句都能被预处理。预处理语句 API 适用于以下语句：CREATE TABLE、DELETE、DO、INSERT、REPLACE、SELECT、SET、UPDATE 语句和绝大多数 SHOW 语句的变体。从 MySQL 5.1 版开始，预处理机制所支持的语句种类增加了很多，最新的语句清单可以在 MySQL 5.1 版的《MySQL 参考手册》里查到。

要使用二进制协议，必须创建一个语句处理程序。有了这个处理程序，就可以把一条语句发送到服务器去接受预处理了。服务器将分析该语句，记住它，再把关于它的信息发送回客户端库保存到相应的语句处理程序，其他的语句处理会用到这个处理程序。

将要预处理的语句可以有一个或多个用问号（？）表示的“参数”，当执行预处理语句时，你需要把那些参数替换为相应的数据值。比如说，你可以像下面这样对一条语句进行预处理：

```
INSERT INTO score (event_id,student_id,score) VALUES(?,?,?)
```

在这条语句里有 3 个“？”字符，它们的作用是充当参数标记或占位符。等你执行这条语句时，需要提供 3 个数据值绑定到这些占位符上，那些数据值将使这条语句成为一条可以实际执行的完备的语句。预处理语句的这种“参数传递”机制使它们可以重复使用：只要把各有关“参数”替换为一些不同的值，就可以反复多次地执行同一条语句。这意味着某给定语句的文本只需发送一次，等你以后每次执行该语句时，只要发送数据值就行了。这大大改善了重复语句的执行性能，如下所示。

- 服务器只需对语句进行一次分析，而不是每执行一次分析一次。
- 网络开销降低了，因为每次执行时只需发送数据值而不是整个语句。
- 被发送的数据值无需转换为字符串形式，这降低了执行开销。比如说，上面那条 INSERT 语句里的 3 个数据列全都是 INT 数据列。如果使用 mysql_query()或 mysql_real_query()函数去执行一条类似的 INSERT 语句，必须先把 3 个数据值转换为字符串并把它们嵌入到该语句文本里。有了预处理语句接口，只需发送二进制格式的数据值就足够了。
- 对检索到的数据也不必进行转换。在预处理语句的结果集里，非字符串值都将以二进制格式返回，无需转换为字符串形式。

与最初的非二进制协议相比，二进制协议也有如下的一些缺点。

- 它比较难以使用，因为在传输和接收数据值时需要更多的准备工作。
- 二进制协议并不支持所有的语句。例如，USE 语句就不能被预处理。
- 对于交互式程序，最好还是使用原来的非二进制协议。在交互式程序里，从用户那里接收到的每条语句只执行一次，就算使用了预处理语句也不会使性能得到显著改善。只有那些需要反复执行的语句才会在预处理机制下获得最大程度的性能提升。

使用预处理语句的基本流程主要包括以下几个步骤。

(1) 调用mysql_stmt_init()函数分配一个语句处理程序。这个函数将返回一个后续步骤要用到的指向处理程序的指针。

(2) 调用 mysql_stmt_prepare()函数把一条语句发送到服务器去接受预处理并与语句处理程序关联。服务器将分析该语句的某些特征，如语句类型、它包含多少参数标记、它在执行时是否会生成一个结果集，等等。

(3) 如果该语句包含占位符，你在执行它之前必须为每个占位符提供数据，具体做法是为每个参数创建一个MYSQL_BIND结构。每个结构表明一个参数的数据类型、它的值、它是不是NULL，等等。然后通过调用mysql_stmt_bind_param()函数把这些结构绑定到该语句上。

(4) 调用mysql_stmt_execute()函数执行该语句。

(5) 如果该语句（如 INSERT 或 UPDATE 语句）只修改数据而不生成结果集，调用 mysql_stmt_affected_rows()函数确定该语句影响了多少数据行。

(6) 如果该语句会生成结果集，调用mysql_stmt_result_metadata()函数获得关于该结果集的元数据。结果集里的数据行需要使用MYSQL_BIND结构来取回，但这次是用它们接收从服务器返回的数据而不是向服务器发送数据。你必须为结果集里的每个数据列分别创建一个MYSQL_BIND结构，它们包含关于你期望从服务器收到的每个数据行的数据值的信息。调用 mysql_stmt_bind_result()函数把这些结构绑定到你的语句处理程序上。接下来，通过反复调用mysql_stmt_fetch()函数依次取回每个数据行。在取回一个数据行之后，你就可以访问当前数据行里的数据列值了。

作为一个可选操作，还可以在调用 mysql_stmt_fetch()函数之前调用 mysql_stmt_store_result()函数，这将把结果集里的数据行一次性地从服务器全部取回并缓冲到客户端的内存里。也可以通过调用 mysql_stmt_num_rows()函数查知结果集里的数据行的个数，如果结果集为空，mysql_stmt_num_ rows()函数调用将返回零。

在取回结果集之后，要记得调用mysql_stmt_freeresult()函数释放与之相关的内存。

(7) 如果你想再次执行刚才的语句，返回第(4)步。

(8) 如果你想使用这个处理程序对另一条语句进行预处理，返回第(2)步。

(9) 在用完语句处理程序之后，别忘了调用mysql_stmt_close()函数释放它。如果某个客户连接关闭时服务器里还有与该连接相关的预处理语句，服务器将自动释放它们。

在同一个客户程序里，你可以先准备好多条语句，然后按照合适的顺序依次执行它们。

接下来，我们将编写一个简单的程序把一些数据行插入一个数据表，然后再检索它们。处理INSERT 语句的部分演示了如何在语句里使用占位符、如何在执行预处理语句时把数据值传递给服务器与预处理语句绑定；处理SELECT语句的部分演示了如何对预处理语句的结果集进行检索。在sampdb发行版本的capi目录里的prepared.c和process_prepared_statement.c文件里可以找到源代码。这里省略了用来建立连接的代码，因为它们与前几个示例程序里的大同小异。

这个程序的重点是为了使用预处理语句而进行的准备工作，下面是相关代码：

```
void
process_prepared_statements (MYSQL *conn)
{
MYSQL_STMT *stmt;
char       *use_stmt = "USE sampdb";
char       *drop_stmt = "DROP TABLE IF EXISTS t";
char       *create_stmt =
  "CREATE TABLE t (i INT, f FLOAT, c CHAR(24), dt DATETIME)";
```

```
  /* select database and create test table */

  if (mysql_query (conn, use_stmt) != 0
    || mysql_query (conn, drop_stmt) != 0
    || mysql_query (conn, create_stmt) != 0)
  {
    print_error (conn, "Could not set up test table");
    return;
  }

  stmt = mysql_stmt_init (conn);  /* allocate statement handler */
  if (stmt == NULL)
  {
    print_error (conn, "Could not initialize statement handler");
    return;
  }

  /* insert and retrieve some records */
  insert_rows (stmt);
  select_rows (stmt);

  mysql_stmt_close (stmt);       /* deallocate statement handler */
}
```

首先，我们选择了一个数据库并创建了一个测试数据表。这个数据表包含4个不同数据类型的数据列：一个INT、一个FLOAT、一个CHAR和一个DATETIME。你将看到，这几种数据类型要区别对待。

在创建测试数据表之后，调用mysql_stmt_init()函数分配一个预处理语句处理程序，然后插入并检索一些数据行，最后释放处理程序。所有的实际工作发生在insert_rows()和select_rows()函数里，我们马上就要讲到它们。在出错处理部分，这个程序还使用了一个名为print_stmt_error()的函数，它与前几个示例程序使用的print_error()函数很相似，但它将调用特定于预处理语句的出错报告函数：

```
static void
print_stmt_error (MYSQL_STMT *stmt, char *message)
{
  fprintf (stderr, "%s\n", message);
  if (stmt != NULL)
  {
    fprintf (stderr, "Error %u (%s): %s\n",
             mysql_stmt_errno (stmt),
             mysql_stmt_sqlstate (stmt),
             mysql_stmt_error (stmt));
  }
}
```

insert_rows()函数负责把一些新数据行添加到测试数据表里：

```
static void
insert_rows (MYSQL_STMT *stmt)
{
char           *stmt_str = "INSERT INTO t (i,f,c,dt) VALUES(?,?,?,?)";
MYSQL_BIND     param[4];
```

```
int           my_int;
float         my_float;
char          my_str[26]; /* ctime() returns 26-character string */
MYSQL_TIME    my_datetime;
unsigned long my_str_length;
time_t        clock;
struct tm     *cur_time;
int           i;

  printf ("Inserting records...\n");

  if (mysql_stmt_prepare (stmt, stmt_str, strlen (stmt_str)) != 0)
  {
    print_stmt_error (stmt, "Could not prepare INSERT statement");
    return;
  }

  /*
   * zero the parameter structures, and then perform all parameter
   * initialization that is constant and does not change for each row
   */

  memset ((void *) param, 0, sizeof (param));

  /* set up INT parameter */

  param[0].buffer_type = MYSQL_TYPE_LONG;
  param[0].buffer = (void *) &my_int;
  param[0].is_unsigned = 0;
  param[0].is_null = 0;
  /* buffer_length, length need not be set */

  /* set up FLOAT parameter */

  param[1].buffer_type = MYSQL_TYPE_FLOAT;
  param[1].buffer = (void *) &my_float;
  param[1].is_null = 0;
  /* is_unsigned, buffer_length, length need not be set */

  /* set up CHAR parameter */

  param[2].buffer_type = MYSQL_TYPE_STRING;
  param[2].buffer = (void *) my_str;
  param[2].buffer_length = sizeof (my_str);
  param[2].is_null = 0;
  /* is_unsigned need not be set, length is set later */

  /* set up DATETIME parameter */

  param[3].buffer_type = MYSQL_TYPE_DATETIME;
  param[3].buffer = (void *) &my_datetime;
  param[3].is_null = 0;
  /* is_unsigned, buffer_length, length need not be set */
```

```
  if (mysql_stmt_bind_param (stmt, param) != 0)
  {
    print_stmt_error (stmt, "Could not bind parameters for INSERT");
    return;
  }

  for (i = 1; i <= 5; i++)
  {
    printf ("Inserting record %d...\n", i);

    (void) time (&clock); /* get current time */

    /* set the variables that are associated with each parameter */

    /* param[0]: set my_int value */
    my_int = i;

    /* param[1]: set my_float value */
    my_float = (float) i;

    /* param[2]: set my_str to current ctime() string value */
    /* and set length to point to var that indicates my_str length */
    (void) strcpy (my_str, ctime (&clock));
    my_str[24] = '\0';  /* chop off trailing newline */
    my_str_length = strlen (my_str);
    param[2].length = &my_str_length;

    /* param[3]: set my_datetime to current date and time components */
    cur_time = localtime (&clock);
    my_datetime.year = cur_time->tm_year + 1900;
    my_datetime.month = cur_time->tm_mon + 1;
    my_datetime.day = cur_time->tm_mday;
    my_datetime.hour = cur_time->tm_hour;
    my_datetime.minute = cur_time->tm_min;
    my_datetime.second = cur_time->tm_sec;
    my_datetime.second_part = 0;
    my_datetime.neg = 0;

    if (mysql_stmt_execute (stmt) != 0)
    {
      print_stmt_error (stmt, "Could not execute statement");
      return;
    }

    sleep (1);  /* pause briefly (to let the time change) */
  }
}
```

insert_rows()函数的目的是把 5 个数据行插入到测试数据表里，它们都包含以下这些值。

- 一个小于 5 大于 1 的 INT 值。
- 一个小于 5.0 大于 1.0 的 FLOAT 值。
- 一个 CHAR 值。为了生成这些值，我们将调用 ctime()系统函数来获得“当前时间”值的字符串形式。ctime()函数的返回值的格式如下所示：

```
Sun Sep 19 16:47:23 CDT 2004
```

- 一个 DATETIME 值。这也将是“当前时间”值，但被保存在一个 MYSQL_TIME 结构里。二进制协议使用 MYSQL_TIME 结构来传输 DATETIME、TIMESTAMP、DATE 和 TIME 值。

insert_rows()函数做的第一件事是把下面的INSERT语句传递给mysql_stmt_prepare()函数：

```
INSERT INTO t (i,f,c,dt) VALUES(?,?,?,?)
```

这条语句包含 4 个占位符，所以每次执行这条语句时需要提供 4 个数据值。占位符通常代表 VALUES()列表里或 WHERE 子句里的数据值。但有些地方不允许使用占位符，如下所示。

- 作为数据表名或数据列名等标识符的占位符。比如说，下面这条语句就是非法的：

```
SELECT * FROM ?
```

- 你可以在操作符的任何一侧使用占位符，但不允许同时在操作符的两侧使用占位符。下面这条语句是合法的：

```
SELECT * FROM student WHERE student_id = ?
```

但下面这条语句是非法的：

```
SELECT * FROM student WHERE ? = ?
```

这条限制很有必要，这样才能让服务器确定参数的数据类型。

下一步是创建一个以 MYSQL_BIND 结构为元素的数组，其中每个元素对应一个占位符。具体到这里的 insert_rows()函数，数组的设置分两个阶段完成。

(1) 在每个 MYSQL_BIND 结构里，对所有在插入新数据行时不会发生变化的部分进行初始化。

(2) 执行用来插入新数据行的循环，在插入一个新数据行之前，对每个 MYSQL_BIND 结构里所有需要改变的部分进行初始化。

也可以把初始化工作全部安排在循环体内进行，但那么做的效率要低一些。

在初始化的第一阶段，首先对包含 MYSQL_BIND 结构的 param 数组的内容清零。这个程序使用的是 memset()函数，如果你的系统不支持 memset()函数，也可以用 bzero()函数。下面两条语句是等价的：

```
memset ((void *) param, 0, sizeof (param));
bzero ((void *) param, sizeof (param));
```

对 param 数组进行清零将隐含地把所有的结构成员设置为零。接下来的代码把一些成员设置为零以明确地表明正在发生的事情，但那几行代码并不是必不可少的。在实际工作中，如果对某个结构进行了清零，就没有必要把该结构的任何成员赋值为零了。

下一步是把正确的信息赋值给 MYSQL_BIND 数组里的每个参数。对于每个参数，需要设置的结构成员取决于将传输的值的数据类型，如下所示。

- buffer_type 成员是必须设置的，它用来表明参数值的数据类型。附录 G（需要上网查阅）里有一个表格完整地列出了允许使用的数据类型代码，以及与每个代码相对应的 SQL 数据类型和 C 数据类型。
- buffer 成员应该被设置为用来保存数据值的变量的地址。insert_rows()函数声明了 4 个变量来保存数据行值：my_int、my_float、my_str 和 my_datetime。每个 param[i].buffer 值被设置为指向相应的变量。在插入新数据行时，我们将把这 4 个变量设置为数据表列的值并用它们创建新数据行。

- is_unsigned 成员只适用于整数数据类型。它的可取值是 true（非零）或 false（零），表明参数是否对应着一个 UNSIGNED 整数。数据表包含一个带正负号的 INT 数据列，所以要把 is_unsigned 成员设置为零。假如数据列是 INT UNSIGNED 类型，我们将需要把 is_unsigned 成员设置为 1，同时还需要把 my_int 变量声明为 unsigned int 类型而不是 int 类型。
- is_null 成员表明你是否正在传输 NULL 值。一般来说，你应该把这个成员设置为 my_bool 变量的地址。然后，在插入任何一个给定数据行之前，把这个变量设置为 true 或 false 以表明将被插入的值是否为 NULL。但如果根本不允许发送 NULL 值（就像本例一样），你可以把 is_null 设置为零，并且不用声明 my_bool 变量。
- 对于字符串值或二进制数据（BLOB 值），需要多设置两个 MYSQL_BIND 成员：一个负责给出将被用来容纳字符串值的缓冲区的长度，另一个用来给出正在被传输的当前值的实际长度。在许多时候，这两个成员的值是一样的，但如果你使用了一个固定长度的缓冲区而被发送的值的长度会随着数据行的不同而变化，它们的值将会不同。buffer_length 成员用来给出缓冲区的长度。length 是一个指针，它应该被设置为一个 unsigned long 变量的地址，该变量包含被传输的值的实际长度。对于数值类型和时间数据类型，buffer_length 和 length 用不着设置，这些类型的长度都是固定不变的，因而可以根据 buffer_type 的值确定下来。比如说，MYSQL_TYPE_LONG 和 MYSQL_TYPE_FLOAT 对应着长度是 4 个字节的值。

完成了对 MYSQL_BIND 数组的初始化之后，把这个数组传递到 mysql_stmt_bind_param() 函数，这将把这个数组绑定到预处理语句上。接下来，我们还需要对各 MYSQL_BIND 结构所指向的变量赋值，然后才能执行语句。这发生在一个将执行 5 遍的循环里。该循环的每次迭代将把相应的值赋给语句的参数。

- 对于整数和浮点参数，只有对相关的 int 和 float 变量进行赋值是必要的。
- 对于字符串参数，需要把字符串格式的当前时间赋给相关的 char 变量。这个值是通过调用 ctime() 函数、然后去掉换行字符而获得的。
- datetime 参数也将被赋值为当前时间，但这是通过把时间值的各部分分别赋值给相关 MYSQL_BIND 结构的各成员而做到的。

把参数值全部设置好以后，调用 mysql_stmt_execute() 函数来执行语句。这个函数将把当前值传递到服务器，服务器将把它们安插到预处理语句中的相应位置并执行该语句。

当 insert_rows() 函数返回时，测试数据表已经被填充好了。我们可以调用 select_rows() 函数去检索它们：

```
static void
select_rows (MYSQL_STMT *stmt)
{
char          *stmt_str = "SELECT i, f, c, dt FROM t";
MYSQL_BIND    param[4];
int           my_int;
float         my_float;
char          my_str[24];
unsigned long my_str_length;
MYSQL_TIME    my_datetime;
my_bool       is_null[4];
```

```
printf ("Retrieving records...\n");

if (mysql_stmt_prepare (stmt, stmt_str, strlen (stmt_str)) != 0)
{
  print_stmt_error (stmt, "Could not prepare SELECT statement");
  return;
}

if (mysql_stmt_field_count (stmt) != 4)
{
  print_stmt_error (stmt, "Unexpected column count from SELECT");
  return;
}

/*
 * initialize the result column structures
 */

memset ((void *) param, 0, sizeof (param)); /* zero the structures */

/* set up INT parameter */

param[0].buffer_type = MYSQL_TYPE_LONG;
param[0].buffer = (void *) &my_int;
param[0].is_unsigned = 0;
param[0].is_null = &is_null[0];
/* buffer_length, length need not be set */

/* set up FLOAT parameter */

param[1].buffer_type = MYSQL_TYPE_FLOAT;
param[1].buffer = (void *) &my_float;
param[1].is_null = &is_null[1];
/* is_unsigned, buffer_length, length need not be set */

/* set up CHAR parameter */

param[2].buffer_type = MYSQL_TYPE_STRING;
param[2].buffer = (void *) my_str;
param[2].buffer_length = sizeof (my_str);
param[2].length = &my_str_length;
param[2].is_null = &is_null[2];
/* is_unsigned need not be set */

/* set up DATETIME parameter */

param[3].buffer_type = MYSQL_TYPE_DATETIME;
param[3].buffer = (void *) &my_datetime;
param[3].is_null = &is_null[3];
/* is_unsigned, buffer_length, length need not be set */

if (mysql_stmt_bind_result (stmt, param) != 0)
{
  print_stmt_error (stmt, "Could not bind parameters for SELECT");
```

```
    return;
  }

  if (mysql_stmt_execute (stmt) != 0)
  {
    print_stmt_error (stmt, "Could not execute SELECT");
    return;
  }

  /*
   * fetch result set into client memory; this is optional, but it
   * allows mysql_stmt_num_rows() to be called to determine the
   * number of rows in the result set.
   */

  if (mysql_stmt_store_result (stmt) != 0)
  {
    print_stmt_error (stmt, "Could not buffer result set");
    return;
  }
  else
  {
    /* mysql_stmt_store_result() makes row count available */
    printf ("Number of rows retrieved: %lu\n",
            (unsigned long) mysql_stmt_num_rows (stmt));
  }

  while (mysql_stmt_fetch (stmt) == 0)  /* fetch each row */
  {
    /* display row values */
    printf ("%d  ", my_int);
    printf ("%.2f  ", my_float);
    printf ("%*.*s  ", my_str_length, my_str_length, my_str);
    printf ("%04d-%02d-%02d %02d:%02d:%02d\n",
            my_datetime.year,
            my_datetime.month,
            my_datetime.day,
            my_datetime.hour,
            my_datetime.minute,
            my_datetime.second);
  }

  mysql_stmt_free_result (stmt);      /* deallocate result set */
}
```

select_rows()函数由 3 个部分组成：对一条 SELECT 语句进行预处理，执行该语句，对该语句的执行结果进行检索。本例中， SELECT 语句不包含任何占位符：

```
SELECT i, f, c, dt FROM t
```

这意味着在执行这条语句之前不需要设置任何 MYSQL_BIND 结构，但也仅此而已。select_rows()函数的重点是像 insert_rows()函数那样，创建一个以 MYSQL_BIND 结构为元素的数组。只不过这次是在执行语句之后用它们去接收来自服务器的数据值，而不是在执行语句之前设置将被发送到服务器

的数据值。

尽管如此，在 select_rows()函数里设置 MYSQL_BIND 数组的过程与刚才在 insert_rows()函数里使用的差不多，均为如下所示。

(1) 对数组进行清零。

(2) 把每个参数的 buffer_type 成员设置为相应的类型代码。

(3) 把每个参数的 buffer 成员设置为指向一个变量；在取回数据行时，相应的数据列值将被存入该变量。

(4) 把整数参数的 is_unsigned 成员设置为零。

(5) 对于字符串参数，把 buffer_length 成员设置为将取回的字节的最大个数，把 length 成员设置为一个 unsigned long 变量的地址。在取回数据行时，这个变量将被设置为实际取回的字节个数。

(6) 对于每一个参数，把 is_null 成员设置为一个 my_bool 变量的地址。在取回数据行时，这些变量将被设置为一个值，它用来表明取回的值是不是 NULL。（对于本例，我们在取回数据行之后将忽略这些变量，因为我们早就知道测试数据表不包含 NULL 值。但一般来说，还是应该检查它们。）

在设置好参数之后，通过调用 mysql_stmt_bind_result()函数把这个数组绑定到语句上，然后执行该语句。

至此，我们已经可以调用 mysql_stmt_fetch()函数来取回数据行了，但示例采用了另一种做法：先调用 mysql_stmt_store_result()函数取回整个结果集并把它缓冲到客户端的内存里。这么做的好处是你可以调用 mysql_stmt_num_rows()函数查出结果集里有多少个数据行，坏处是需要占用更多的客户端内存。

用来取回数据行的循环反复调用 mysql_stmt_fetch()函数直到它返回一个非零值为止。在每次取回之后，与参数结构相关联的变量将包含当前数据行的各数据列的值。

在把所有数据行全部取回之后，调用 mysql_stmt_free_result()来释放与结果集相关联的所有内存。

select_rows()函数返回之后，它调用了 mysql_stmt_close()函数来释放预处理语句处理程序。

前面全面介绍了预处理语句接口和一些关键的函数。客户端库还提供了其他几个相关的函数，关于它们的更多信息请参阅附录 G（需上网查阅）。

第 8 章

使用 Perl DBI 编写 MySQL 程序

本章将介绍 MySQL 的 Perl 语言编程接口 DBI 的使用方法。如果你想了解 DBI 的工作原理和体系结构（尤其是 DBI 与 C API 和 PHP API 的对比），请参见第 6 章。

本书的第 1 章创建了一个示例数据库 sampdb，并在其中为考试记分项目和美国历史研究会分别创建了一些数据表。本章将以这个示例数据库为例来展开讨论。为了更好地学习本章，你最好对 Perl 有一定的了解。虽说不熟悉 Perl 也完全能通过复制本章中的示例代码编写出一些脚本来，但买一本好的 Perl 图书却肯定是一项非常划算的投资。由 Wall、Christinsen 和 Orwant 撰写的 *Programming Perl, Third Edition*（O'Reilly，2000 年）就是一本这样的好书。

DBI 的当前版本是 1.601，但本章中的大部分讨论也同样适用于较早的版本。DBI 要求运行在 Perl 5.6.0（5.6.1 优先于 5.6.0）或更高的版本上。必须安装 DBD::mysql Perl 模块、MySQL C 客户端库和头文件。如果你还想利用本章介绍的技巧编写一些基于 Web 的 DBI 脚本，就应该安装 CGI.pm 模块。在本章中，CGI.pm 模块将与 Web 服务器 Apache 配合使用。如果你需要这几个软件，请参阅附录 A。从附录 A 也可查知如何获得本章的示例脚本，它们是 sampdb 发行版本的组成部分，如果下载了 sampdb 发行版本，就用不着自己动手输入脚本了。本章中用到的脚本都放在 sampdb 发行版本的 perlapi 目录里。

8

限于篇幅，本章只介绍了讨论涉及的那些 Perl DBI 方法和变量。附录 H（需上网查阅）介绍了许多 Perl DBI 方法和变量。如果你在今后需要用到其他 DBI 方法或变量，可以把附录 H 当做参考手册。在线文档可以用下面的命令查阅到：

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

在数据库驱动（database driver，DBD）层次上，MySQL 驱动是在 MySQL C 客户端库的基础构建出来的，所以它们有很多相似的地方。对客户端库的详细介绍请参见第 7 章。

8.1 Perl 脚本的特点

Perl 脚本是文本文件，可以用任意一种文本编辑器来创建。本章的 Perl 脚本都遵守 Unix 系统上的脚本编写规定：在第一行用“#!”给出一个路径名，路径名上的程序将被用来执行这个脚本。我在脚本的第一行是这样写的：

```
#!/usr/bin/perl
```

在 Unix 系统，如果 Perl 在你系统上的路径名不同，如/usr/local/bin/perl5 或/opt/bin/perl，那么需要修改#!行。否则，Perl 脚本将无法在你的系统上正确执行。

你可以像下面这样在任何系统上调用 Perl 脚本 myscript.pl 来运行：

```
% perl myscript.pl
```

不过，不必写出 perl 程序也能执行脚本，在 Unix 上可以用 chmod 命令修改文件模式，使脚本可执行，如下所示：

```
% chmod +x myscript.pl
```

以后，只需输入脚本的名字就能执行它了，如下所示：

```
% ./myscript.pl
```

本章的示例都将采用这种样式执行 Perl 脚本。如果脚本文件保存在你的当前目录（以符号“.”表示）里而你的 shell 却没有把当前目录添加到搜索路径里，脚本文件名的前面就必须有“./”记号。否则，脚本文件名前面的“./”记号就可以省略：

```
% myscript.pl
```

在 Windows 系统上，你可以在 Perl 和以“.pl”结尾的文件名之间建立一个文件名关联。就拿 ActiveState Perl 来说吧，它的安装程序允许你建立一个文件名关联，让 Perl 去负责执行以“.pl”结尾的文件。此时，只需在命令行上给出 Perl 脚本的名字就能执行它们了：

```
C:\> myscript.pl
```

8.2　Perl DBI 概述

本节介绍了 DBI 的背景知识，这些知识是你在编写自己的脚本和阅读别人写的脚本时必须掌握的。如果你已经对 DBI 比较熟悉，可以直接跳到 8.3 节。

8.2.1　DBI数据类型

Perl DBI API 与第 7 章介绍的 C 客户端库的用法有很多相似的地方。在使用 C 客户端库时，需要调用各种函数，通过指向结构或数组的指针来访问与 MySQL 有关的数据。在使用 DBI API 时，还得用到函数和指向结构的指针，只是它们的称谓都发生了变化：函数现在被称为“方法”（method）、指针被称为“引用”（reference）、指针变量被称为“句柄”（handle），句柄指向的结构被称为“对象”（object）。

DBI 句柄可以细分为很多种，它们在 DBI 文档里大都有约定俗成的称呼，如表 8-1 所示。这些句柄的用法将随着我们学习的深入而越来越清晰。有些常用的非句柄变量也有约定俗成的称呼（见表 8-2）。本章其实用不到这么多变量，但当你阅读别人写的 DBI 脚本时，知道这些变量的含义可就有用了。

表8-1　Perl DBI句柄变量的名称

名　称	含　义
$dbh	指向数据库对象的句柄
$sth	指向语句（查询）的句柄
$fh	指向已打开文件的句柄
$h	“普通”的句柄，它的含义要由上下文来决定

表8-2 常用Perl DBI非句柄变量名

名 字	含 义
$rc	会返回true或false的操作的返回代码
$rv	会返回一个整数的操作的返回值
$rows	会返回数据行计数值的操作的返回值
$ary	一个数组（列表），代表着查询命令所返回的一个数据行

8.2.2 一个简单的DBI脚本

我们就从一个简单的脚本 dump_members.pl 开始吧。这个脚本演示了 DBI 程序设计中的几个标准概念，如与 MySQL 服务器的连接和断开、发出 SQL 语句、检索数据等。这个脚本将以制表符分隔格式输出一份美国历史研究会的会员名单。可我们现在关心的并不是它的输出格式是否美观，眼下最重要的是弄明白如何使用 DBI 脚本。dump_members.pl 如下所示：

下面是 dump_members.pl 脚本的源代码：

```
#!/usr/bin/perl
# dump_members.pl - dump Historical League's membership list

use strict;
use warnings;
use DBI;

# data source name, username, password, connection attributes
my $dsn = "DBI:mysql:sampdb:localhost";
my $user_name = "sampadm";
my $password = "secret";
my %conn_attrs = (RaiseError => 1, PrintError => 0, AutoCommit => 1);

# connect to database
my $dbh = DBI->connect ($dsn, $user_name, $password, \%conn_attrs);

# issue query
my $sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
        . " street, city, state, zip, phone FROM member ORDER BY last_name");
$sth->execute ();

# read and display query result
while (my @ary = $sth->fetchrow_array ())
{
  print join ("\t", @ary), "\n";
}
$sth->finish ();

$dbh->disconnect ();
```

如果你想试用一下这个脚本，可以用它在 sampdb 发行版本里的副本，也可以用一个文本编辑器来亲手创建它。（如果你使用的是文字处理器程序，一定要把脚本保存为普通文本格式，而不要保存为文字处理器的默认格式。）另外，你可能至少要修改一些连接参数，如主机名、数据库名、用户名、口令等。（本章还有其他一些脚本也用到了连接参数，它们也需要做适当的修改。）在稍后的 8.2.9 节

里，我们将学习如何从选项文件里读出这些参数而不是把它们直接放在脚本里。

下面，对这个脚本逐行分析。它在第一行给出了一个标准的 Perl 路径：

```
#!/usr/bin/perl
```

这一行是本章中每个脚本都有的内容，以后将不再重复说明了。

在脚本里对它的功能做一下介绍是一种良好的编程习惯，它能帮助别人了解你脚本的用途，接下来的一行就是一条这样的注释：

```
# dump_members.pl - dump Historical League's membership list
```

从“#”字符到行尾之间的内容将被认为是一条注释。用注释对脚本程序的各种过程作出解释是一种良好的编程习惯。

接下来是几条 use 语句：

```
use strict;
use warnings;
use DBI;
```

use strict 将使 Perl 要求你在使用变量之前必须先声明。在编写脚本时，可以不使用这条 use strict 命令，但写上它有助于捕获错误，所以我建议大家在自己的脚本里加上这一行。比如说，如果你声明了一个变量$my_var 但在后面的代码里把它错写为$mv_var，在严格模式下运行脚本就会出现出错信息：

```
Global symbol "$mv_var" requires explicit package name at line n
```

当你看到这条出错信息时，你就会想：“怎么回事？我从没用过名为$mv_var 的变量呀！”于是，你会去检查脚本程序的第 n 行，发现自己把$my_var 错写为$mv_var 了，于是改正过来。如果不是在严格模式下，Perl 就不会认为$mv_var 是一个错误，它将默默创建出一个名为$mv_var 的新变量并赋值为 undef（意思是“未定义”），而你则会疑惑为什么脚本没有达到预定的效果。

use warnings 的作用是让 Perl 在它发现程序员使用了有疑问的语言结构或是执行了有问题的操作（比如说，试图输出一个尚未经过初始化的变量）时发出一条警告。这很有用，因为它可以警示你哪些代码应该重写。

use DBI 告知 Perl 解释器，需要加载 DBI 模块。要是缺少了这一行，DBI 脚本就将无法正常工作。你用不着细致地指定将要使用哪一个 DBD 级的驱动模块，因为 DBI 会在你连接数据库时激活正确的驱动模块。

因为这个脚本将运行在严格模式下，所以我们必须用关键字 my 声明将要用到的每一个变量。你可以把它想象成脚本程序在说：“我明确地把这些东西声明为我的变量。”接下来的脚本首先设置连接参数变量，然后用它们连接数据库：

```
# data source name, username, password, connection attributes
my $dsn = "DBI:mysql:sampdb:localhost";
my $user_name = "sampadm";
my $password = "secret";
my %conn_attrs = (RaiseError => 1, PrintError => 0, AutoCommit => 1);

# connect to database
my $dbh = DBI->connect ($dsn, $user_name, $password, \%conn_attrs);
```

connect()是 DBI 类的一个方法，所以要用 DBI->connect()的形式来调用。你用不着追究它到

底是什么意思，它只是面向对象的语言用来调用类方法的一种手段而已。（如果你真想知道，告诉你吧，它表明 connect()是一个“属于”DBI 的函数。）connect()需要以下几个参数。

- 数据源，也叫数据源名，即 DSN。DSN 指定使用的是哪个 DBD 模块，也可能指定其他参数。
- MySQL 账户的用户名和口令。
- 一个可选参数，用来给出额外的连接属性。如果需要用到这个参数，它必须是对散列的引用，连接属性的名字及其设置值就存放在其中。

数据源格式根据你要使用的 DBI 模型的需求确定。对于 MySQL 驱动器，允许的 DNS 格式下列两种：

```
DBI:mysql:db_name
DBI:mysql:db_name:host_name
```

DBI 的大小写无关紧要，但 mysql 却必须是小写。db_name 是你准备使用的数据库的名字，host_name 是 MySQL 服务器在其上运行的主机的名字。如果你没有给出主机名，它的默认值将是 localhost。（8.2.9 节介绍了其他几种数据源格式。）

具体到这个例子，指针%conn_attrs 所指向的连接属性散列将启用 RaiseError 属性，禁用 PrintError 属性。这些设置将使 DBI 去检查是否发生了与数据库有关的错误，如果检测到错误就退出执行，但不会输出一条出错消息。（这也正是在 dump_members.pl 脚本里看不到任何出错检查代码的原因——全都由 DBI 代劳了。）8.2.3 节将介绍一些其他的出错响应办法。

属性散列还启用了 AutoCommit 属性。这在现阶段并不是必需的，它的好处是可以明确地表明这个脚本将在自动提交模式下进行事务处理。这个脚本里没有任何显而易见的事务，但 DBI 在不久的将来有可能会要求脚本必须对 AutoCommit 属性做出明确的设定。现在就在编写脚本时这么做不失为一种未雨绸缪的好办法。

在设定连接属性的时候，还可以直接把那个散列写在 connect()函数调用里，如下所示：

```
my $dbh = DBI->connect ($dsn, $user_name, $password,
                        { RaiseError => 1, PrintError => 0, AutoCommit => 1 });
```

上面两种编程风格在实际操作上没有任何差别，不同的人可以随意选择自认为更容易阅读和编辑的做法。

如果 connect()调用成功，它将返回一个数据库句柄，我们再把这个句柄赋值给变量$dbh。在默认的情况下，如果调用失败，connect()将返回 undef。但因为这个脚本启用了 RaiseError 属性，所以如果真的在 connect()的调用过程中出现了错误，DBI 就会在显示一条出错信息后退出执行。（其他 DBI 方法也是如此，我很快就会告诉大家它们在执行出错时会返回什么值。但如果启用了 RaiseError 属性，它们将不会返回。）

在连接上数据库之后，dump_members.pl 将发出一条 SELECT 语句去检索“美国历史研究会”的会员名单，然后再执行一个循环语句以处理它检索到的每一个数据行。这些数据行构成了结果集。为了进行这次 SELECT 查询，应先准备好语句，然后再执行它：

```
# issue query
my $sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
        . " street, city, state, zip, phone FROM member ORDER BY last_name");
$sth->execute ();
```

我们用数据库句柄调用了 prepare()方法，它将把 SQL 语句传递给驱动模块去进行执行前的预

处理。有些驱动模块的确会在这一阶段对语句作一些处理，另一些则只是简单地记住这条语句并等你用 execute()方法去执行它们。prepare()方法的返回值是一个语句句柄，我们把它赋值给了$sth。此后，只要是与这个语句有关的处理工作都要通过这个语句句柄来完成。

注意，语句字符串并不需要用分号来结尾。在与 mysql 程序打了那么长时间的交道之后，你可能已经习惯在 SQL 语句的末尾加上一个分号来结束了。但与 DBI 打交道时，还是把这个习惯改一改比较好，因为分号往往会导致启用因语法错误而执行失败。这句话也适用于启用字符串末尾的\g 或\G 结束符——最好不要加上它们。这些语句结束符只有在 mysql 程序里才是必需的，你通过 DBI 脚本而发出的启用不需要使用它们。在 DBI 脚本里，启用字符串的尾字符就隐含地标志了启用的结束，用不着再启用加上明确的结束符了。

如果你在调用某个方法时没有给它传递任何参数，就可以省略它后面的括号。也就是说，下面两种写法是等价的：

```
$sth->execute ();
$sth->execute;
```

我喜欢留着那些括号，因为它们能帮我把脚本中的方法调用和变量引用区分开。你也许不喜欢。

execute()调用完成之后，我们就可以处理会员名单数据行了。在 dump_members.pl 脚本里，用来提取数据行的循环语句很简单，它只是把每一行的内容依次打印为一行以制表符分隔的值而已：

```
# read and display query result
while (my @ary = $sth->fetchrow_array ())
{
  print join ("\t", @ary), "\n";
}
$sth->finish ();
```

fetchrow_array()的返回值是由当前数据行各数据列的值构成的一个数组，如果已经到达结果集里的最后一个数据行，它就返回一个空数组。于是，这个循环语句将依次取回 SELECT 语句返回的每一个数据行，在数据列之间加上制表符，然后再打印。

数据库里的 NULL 值将被返回为 Perl 脚本里的 undef 值，但它们将被打印为空字符串而不是单词“NULL”。undef 值在脚本运行时还有另外一个效果：它们会导致 Perl 解释器给出如下所示的警告信息：

```
Use of uninitialized value in join at dump_members.pl line n.
```

这些警告信息是因为程序使用了 use warning 而产生的，如果去掉这个语句再次运行脚本，这些警告信息就不会出现了。可是，警告模式有助于发现问题（如打印一个未经初始化的变量），所以我们宁愿保留 use warning 并想办法通过检测和处理 undef 值来消除警告。8.2.5 节将介绍一些解决这一问题的办法。

在 print 语句里，制表符和换行符（分别表示为“\t”和“\n”）都加上了双引号，因为 Perl 只能识别出放在双引号（而非单引号）里的转义序列。如果使用的是单引号，这个脚本就会原封不动地打印出大量的“\t”和“\n”字符来。

提取数据行的循环语句结束后，finish()调用表示不再需要语句句柄，现在可以释放为之分配的各种临时资源。就 dump_members.pl 脚本而言，finish()调用只起到了一个演示的目的。其实用不着调用它，因为提取数据行的调用将在它到达结果集末尾时自动完成这件事情。但如果你只取回结果

集的一部分而没有到达其末尾（例如，因为某种原因仅取回了第一行），finish()就能发挥大作用。在以后的示例里，如果没有必要，将不再使用 finish()调用。

打印完会员名单，我们的工作也就结束了，所以要断开与服务器的连接并退出脚本：

```
$dbh->disconnect ();
```

dump_members.pl 演示了大多数 DBI 程序都会涉及的许多概念，即使你对 DBI 的了解只有这么多，也可以开始编写你自己的 DBI 脚本了。比如说，如果只想写出其他数据表的内容，那你只需修改 prepare()方法里的 SELECT 语句。事实上，如果想了解采用这种技术的程序，可以跳到 8.3 节，其中讨论了如何生成美国历史研究会年会请柬和打印出来的名录。不过，DBI 还提供了许多其他有用的功能。我们将在下一节对其中一些进行更详细的讨论，以便让大家了解如何在 Perl 脚本里完成比运行 SELECT 语句更复杂的任务。

8.2.3 出错处理

在调用 connect()方法时，dump_members.pl 脚本启用了 RaiseError 出错处理属性，一旦执行出错，它就会显示出错信息来结束脚本运行，而不是返回出错代码。也许还有一些其他的出错处理办法，你完全可以不用 DBI 代劳而自己去处理。

为了更好地控制 DBI 的出错处理行为，现在来仔细研究一下 connect()调用的最后一个参数：连接属性散列。

```
# data source name, username, password, connection attributes
my $dsn = "DBI:mysql:sampdb:localhost";
my $user_name = "sampadm";
my $password = "secret";
my %conn_attrs = (RaiseError => 1, PrintError => 0, AutoCommit => 1);

# connect to database
my $dbh = DBI->connect ($dsn, $user_name, $password, \%conn_attrs);
```

与出错处理相关的两个属性是 RaiseError 和 PrintError。

- 如果启用了 RaiseError 属性（即设置为非零值），那么，在某个 DBI 方法里出现错误时，DBI 就会发出异常消息，默认时它会调用 die()方法来显示出错信息并退出。
- 如果启用了 PrintError 属性，那么，在某个 DBI 方法里出现错误时，DBI 就会调用 warn()方法来显示一条出错信息却不会退出，脚本仍能继续往下执行。

在默认情况下，DBI 会禁用 RaiseError 属性，启用 PrintError 属性。此时如果 connect()方法调用失败，DBI 将显示一条出错信息，但仍会继续往下执行。因此，即使省略了 connect()方法的第四个参数，DBI 也能向你提供一种默认的出错处理行为。你就可以这样检查 connect()方法的出错情况：

```
my $dbh = DBI->connect ($dsn, $user_name, $password)
            or exit (1);
```

如果真的出现错误，connect()方法将返回 undef 以表明执行失败，而这个 undef 又将导致 exit()被调用。你用不着打印出错信息，因为 DBI 已经打印了。

如果你想显式地把出错处理属性设置为默认值，就应该把 connect()调用写成这样：

```
my %conn_attrs = (RaiseError => 0, PrintError => 1, AutoCommit => 1);
```

```
my $dbh = DBI->connect ($dsn, $user_name, $password, \%conn_attrs)
            or exit (1);
```

虽说这会让你多打不少字，但它能让别人更清楚地了解你使用的是什么出错处理行为。

如果你想自己检查错误并打印信息，就应该禁用 RaiseError 和 PrintError 属性，如下所示：

```
my %conn_attrs = (RaiseError => 0, PrintError => 0, AutoCommit => 1);
my $dbh = DBI->connect ($dsn, $user_name, $password, \%conn_attrs)
            or die "Could not connect to server: $DBI::err ($DBI::errstr)\n";
```

代码里的$DBI::err 和$DBI::errstr 变量在你构造出错信息时非常有用。它们分别包含 MySQL 出错代码和出错字符串，就像 mysql_errno()和 mysql_error()那样。如果没有发生错误，$DBI::err 将是 0 或 undef，而$DBI::errstr 将是空字符串或 undef。换句话说，如果没有发生错误，这两个变量就都将为假。

如果想让 DBI 替你去进行出错处理，就需要启用 RaiseError 并禁用 PrintError，如下所示：

```
my %conn_attrs = (RaiseError => 1, PrintError => 0, AutoCommit => 1);
my $dbh = DBI->connect ($dsn, $user_name, $password, \%conn_attrs);
```

这是一个最省事的办法，本章的大部分脚本都是这样写的。之所以要在启用 RaiseError 的同时禁用 PrintError，是为了避免把出错信息打印两次。（在某些特定的场合，如果同时启用了这两个属性，DBI 就会因为同一个错误而打印两条出错信息。）

如果你想让脚本退出时清除一些代码，启用 RaiseError 的做法是不可取的，不过，可以通过重新定义$SIG{__DIE__}信号处理程序来达到目的。不启用 RaiseError 的另一个理由是 DBI 打印出来的信息专业性太强，如：

```
disconnect(DBI::db=HASH(0x197aae4)) invalidates 1 active statement. Either
destroy statement handles or call finish on them before disconnecting.
```

这是一条对程序员非常有用的信息，但你并不想让普通用户看到。有时候，由你本人进行出错检查应该更好，因为你可以让使用你脚本的人看到一些更有实际意义的信息。对于这种情况，也可以考虑重新定义$SIG{__DIE__}处理程序，好处是你可以启用 RaiseError 以简化出错处理工作，又能把 DBI 提供的默认出错信息修改为你自己的内容。如果要重新定义$SIG{__DIE__}处理程序，请在执行任何 DBI 调用之前先做好下面的事情：

```
$SIG{__DIE__} = sub { die "Sorry, an error occurred\n"; };
```

你也可以像普通情况那样先定义一个子例程，再利用子例程引用设置为信号处理程序值。如下所示：

```
sub die_handler
{
  die "Sorry, an error occurred\n";
}

$SIG{__DIE__} = \&die_handler;
```

下面这个dump_members2.pl脚本演示了如何编写脚本来自行检查错误并打印信息。dump_members2.pl脚本要处理的语句与dump_members.pl脚本里的一模一样，但它明确地禁用了RaiseError和PrintError属性，然后测试每个DBI调用的结果。一旦执行出错，脚本会在退出之前先调用子例程bail_out()来打印一条信息以及$DBI::err和$DBI::errstr变量的内容：

```
#!/usr/bin/perl
# dump_members2.pl - dump Historical League's membership list

use strict;
use warnings;
use DBI;

# data source name, username, password, connection attributes
my $dsn = "DBI:mysql:sampdb:localhost";
my $user_name = "sampadm";
my $password = "secret";
my %conn_attrs = (RaiseError => 0, PrintError => 0, AutoCommit => 1);

# connect to database
my $dbh = DBI->connect ($dsn, $user_name, $password, \%conn_attrs)
            or bail_out ("Cannot connect to database");

# issue query
my $sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
        . " street, city, state, zip, phone FROM member ORDER BY last_name")
          or bail_out ("Cannot prepare query");
$sth->execute ()
  or bail_out ("Cannot execute query");

# read and display query result
while (my @ary = $sth->fetchrow_array ())
{
  print join ("\t", @ary), "\n";
}
!$DBI::err
  or bail_out ("Error during retrieval");

$dbh->disconnect ()
  or bail_out ("Cannot disconnect from database");

# bail_out() subroutine - print error code and string, and then exit

sub bail_out
{
my $message = shift;

  die "$message\nError $DBI::err ($DBI::errstr)\n";
}
```

bail_out()与我们在第 7 章里学习编写 C 语言程序时使用的 print_error()函数很相似，但 bail_out()会退出执行而不是返回到调用者。bail_out()给我们带来的好处有两个：首先，你不必在每次想打印出错消息时都写出$DBI::err 和$DBI::errstr 变量的值；其次，若把出错信息的打印功能封装在子例程里，只需修改一下这个子例程，就能统一改变整个脚本里的出错信息格式。

dump_members2.pl脚本在用来取回数据行的循环语句后面有一个测试。因为脚本不会在fetchrow_array()调用执行出错时自动退出，所以慎重的做法是安排一个测试来检查循环结束的原

因：因为到达结果集的末尾（正常结束），还是因为在执行过程中发生了错误（非正常结束）。这两种情况都会导致循环结束，但发生错误时脚本的输出将是不完整的，如果不进行出错检查，运行这个脚本的人将根本无法知道已经出了问题！如果你打算自行编写代码来进行出错处理，千万不要忘记对你用来取回数据的那个循环的结果进行测试。

8.2.4　处理修改数据行的语句

相对而言，修改数据行的语句（如 DELETE、INSERT、REPLACE、UPDATE 等）比返回数据行的语句（如 SELECT、DESCRIBE、EXPLAIN、SHOW 等）容易处理一些。对于非 SELECT 语句，可以用数据库句柄把它传递到 do()方法里。do()方法将命令的预处理和执行工作合并为了一个步骤。比如说，下面的脚本将创建一名新成员 Marcia Brown，其会员资格失效日期为 2012 年 6 月 3 日：

```
$rows = $dbh->do ("INSERT INTO member (last_name,first_name,expiration)"
                  . " VALUES('Brown','Marcia','2012-06-03')");
```

do()方法的返回值是受到影响的数据行的个数，若执行出错，则返回 undef；若因为某种原因无法确定受影响的数据行个数，则返回-1。有很多原因都会造成 do()方法执行出错，例如语句本身有错误或者你没有权限访问数据表。对于非 undef 返回值，应该注意没有数据行受影响的情况，do()方法返回的并不是数字 0 而是字符串"0E0"（即 0 在 Perl 语言里的科学计数法表示形式）。在数值上下文里，"0E0"将被求值为 0；但在条件表达式里，它却会被求值为真。这使我们很容易把它与 undef 区别开。假如 do()返回的是数字 0，就很难把“发生了错误（undef）”和“没有数据行受到影响”这两种情况区别开了。下面两个测试都能用来判断执行过程是否出现了错误：

```
if (!defined ($rows))
{
  print "An error occurred\n";
}
if (!$rows)
{
  print "An error occurred\n";
}
```

在数值上下文里，"0E0"将被求值为 0，所以下面这段代码能够根据非 undef 的$rows 值正确地计算并打印出数据行的个数：

```
if (!$rows)
{
  print "An error occurred\n";
}
else
{
  $rows += 0; # force conversion to number if value is "0E0"
  print "Number of rows affected: $rows\n";
}
```

还可以用 printf()函数的%d 格式说明符来打印$rows，这将把字符串"0E0"强制转换为一个数字：

```
if (!$rows)
{
  print "An error occurred\n";
```

```
}
else
{
  printf "Number of rows affected: %d\n", $rows;
}
```

do()等价于 prepare()加 execute()。也就是说，前面那句用 do()方法来处理 INSERT 语句的脚本与下面是等价的：

```
$sth = $dbh->prepare ("INSERT INTO member (last_name,first_name,expiration)"
                    . " VALUES('Brown','Marcia','2012-06-03')");
$rows = $sth->execute ();
```

8.2.5 处理返回结果集的语句

本节将介绍使用循环来取回 SELECT 语句（或者像 SELECT 这样能够返回数据行的其他语句，如 DESCRIBE、EXPLAIN、SHOW 等）中的结果数据行。此外，本节还会介绍如何查知结果集里的数据行个数、在不需要使用循环时如何处理结果集、如何一次性检索整个结果集。

1. 编写取回数据行的循环语句

dump_members.pl 脚本使用了一系列 DBI 方法来检索数据：用 prepare()方法让驱动对命令进行预处理，用 execute()方法开始执行命令，再用 fetchrow_array()方法取回结果集里的每一个数据行。

在对有结果集的任何语句进行处理时，prepare()和 execute()都是相当标准的组成部分。但用来取回数据行的方法有好几种（见表 8-3），fetchrow_array()只是其中之一而已。

表8-3 用来取回数据行的DBI方法

方法名称	返 回 值
fetchrow_array()	元素是数据行的值的数值
fetchrow_arrayref()	对由数据行的值构成的数组的引用
fetch()	与fetchrow_arrayref()相同
fetchrow_hashref()	对由数据行的值构成的数组的引用，键是数据列的名字

下面的示例演示了表 8-3 列出的各个方法的用法。这些示例将循环遍历结果集里的每一个数据行，并把数据列的值以逗号分隔打印出来。这些代码在编写时本来可以更有效率的，但因为这里的目的是演示访问各数据列的值时所使用的语法，而不是追求执行效率，所以我没有那样做。

先来看看 fetchrow_array()的用法：

```
while (my @ary = $sth->fetchrow_array ())
{
  my $delim = "";
  for (my $i = 0; $i < @ary; $i++)
  {
    $ary[$i] = "" if !defined ($ary[$i]); # NULL value?
    print $delim, $ary[$i];
    $delim = ",";
  }
  print "\n";
```

8

```
}
```

每调用一次 fetchrow_array()，它就返回一个由数据行的值组成的数值；若已经到达结果集里的最后一个数据行，则返回一个空数组。内层循环将依次检测各数据列的值是否已经定义，如果没有定义，就把它设置为空字符串。这将把 NULL 值（它们在 DBI 脚本里被表示为 undef）全部转换为空字符串。这项工作看起来好像有点多余——因为不管是 undef 还是空字符串，Perl 打印出的都将是空白。之所以要做这样的测试，是因为如果你的脚本运行在启用了警告的模式下，Perl 就会在你试图打印一个 undef 值的时候给出一条“Use of uninitialized value”（使用了未经初始化的值）警告信息。将 undef 转换为空字符串能消除警告。在本章的很多示例里，你都能看到类似的代码。

如果你想把 undef 打印为另外一个值，如字符串"NULL"，只要稍微修改一下测试部分的 if 代码就可以了，如下所示：

```
while (my @ary = $sth->fetchrow_array ())
{
  my $delim = "";
  for (my $i = 0; $i < @ary; $i++)
  {
    $ary[$i] = "NULL" if !defined ($ary[$i]); # NULL value?
    print $delim, $ary[$i];
    $delim = ",";
  }
  print "\n";
}
```

既然是对值组成的数组进行处理，我们就可以利用 map 来简化代码，把数组里的 undef 元素一次性全部转换过来，如下所示：

```
while (my @ary = $sth->fetchrow_array ())
{
  @ary = map { defined ($_) ? $_ : "NULL" } @ary;
  print join (",", @ary), "\n";
}
```

map 将利用花括号里的表达式处理数组中的每一个元素，最后返回一个由转换结果值构成的数组。

除把 fetchrow_array()的返回值赋值给一个数组变量外，还可以把各数据列的值取到一组标量变量中，而这将使我们能够用更有意义的变量名来代替$ary[0]、$ary[1]等。假设你想把会员的姓名和电子邮件地址检索到相应的变量里去，使用 fetchrow_array()方法可以按如下所示选择和提取数据行：

```
my $sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email"
                         . " FROM member ORDER BY last_name");
$sth->execute ();
while (my ($last_name, $first_name, $suffix, $email)
                        = $sth->fetchrow_array ())
{
  # do something with variables
}
```

当你准备像上面这样使用一组变量时，一定要保证语句中给出数据列名称的顺序与你用来保存数据的那组变量的排列顺序相符合。DBI 并不清楚 SELECT 语句里是按怎样的顺序来列出数据列名称的，所以给变量正确赋值的责任就落在了你的身上。利用一种名为“参数绑定”（见 8.2.7 节）的技巧，我

们还可以使数据列的取值被自动赋值给各个变量。

如果要把值取回到变量里，对变量赋值时一定要多加小心。如果循环是以下面这行代码开头的，它将正确工作：

```
while (my ($val) = $sth->fetchrow->array ()) ...
```

值在列表型上下文中被提取，只有在到达最后一行时，测试才会失败。但如果把测试写成下面这样，就会出一些很奇怪的问题：

```
while (my $val = $sth->fetchrow->array ()) ...
```

这里的区别是，值是在标量上下文中提取的，如果$val 的值恰好是 0、undef 或空字符串，那么即使你还没有到达结果集的最后一行，while 语句也会因其条件表达式被求值为假而结束循环。

fetchrow_arrayref()与 fetchrow_array()很相似，但它返回的不是一个由当前行里的数据列的值构成的数组，而是对该数组的引用，若已经到达结果集里的最后一行，则返回 undef。下面是它的使用方法：

```
while (my $ary_ref = $sth->fetchrow_arrayref ())
{
  my $delim = "";
  for (my $i = 0; $i < @{$ary_ref}; $i++)
  {
    $ary_ref->[$i] = "" if !defined ($ary_ref->[$i]); # NULL value?
    print $delim, $ary_ref->[$i];
    $delim = ",";
  }
  print "\n";
}
```


通过数组引用$ary_ref 来访问数组元素。这有点像对指针进行取消引用，所以应使用$ary_ref->[$i]而不是$ary[$i]。如果想把这个引用转换为一个数组，可以使用@{$ary_ref}结构。

fetchrow_arrayref()不适合用来把变量提取到列表中。比如说，下面的循环是无法正确工作的：

```
while (my ($var1, $var2, $var3, $var4) = @{$sth->fetchrow_arrayref ()})
{
  # do something with variables
}
```

只要 fetchrow_arrayref()真的取回了一个数据行，循环尚可以正确执行。可一旦到达结果集里的最后一行，fetchrow_arrayref()就会返回 undef，而@{undef}却是非法的（它类似于在 C 程序里对 NULL 指针取消引用）。

表 8-3 里的第三个方法 fetchrow_hashref()的使用情况如下所示：

```
while (my $hash_ref = $sth->fetchrow_hashref ())
{
  my $delim = "";
  foreach my $key (keys (%{$hash_ref}))
  {
    $hash_ref->{$key} = "" if !defined ($hash_ref->{$key}); # NULL value?
    print $delim, $hash_ref->{$key};
    $delim = ",";
  }
```

```
  print "\n";
}
```

每调用一次 fetchrow_hashref()，它就返回一个对散列的引用，散列由数据行的值构成，键是数据列的名字；若已经到达结果集里的最后一行，则返回 undef。这种情况下，数据列的值没有特定顺序，因为 Perl 散列的元素就是无序的。好在 DBI 会把各数据列的名称用作散列元素的键，这意味着我们完全能够通过$hash_ref 变量访问该数据列的值。这同时还意味着我们完全可以按任意顺序来提取值而不必理会这些数据列在 SELECT 语句里的检索顺序。比如说，如果你想提取某个会员的姓名和电子邮件地址，只需写出下面的代码就行了：

```
while (my $hash_ref = $sth->fetchrow_hashref ())
{
  my $delim = "";
  foreach my $key ("last_name", "first_name", "suffix", "email")
  {
    $hash_ref->{$key} = "" if !defined ($hash_ref->{$key}); # NULL value?
    print $delim, $hash_ref->{$key};
    $delim = ",";
  }
  print "\n";
}
```

fetchrow_hashref()特别适用于这样的场合：你需要把一个数据行的值传递给一个函数，但这个函数却不必知道数据列在 SELECT 语句里的命名顺序。具体做法是：你使用 fetchrow_hashref()方法检索数据行，再编写一个函数让它通过数据列的名称访问数据行的值。

在使用 fetchrow_hashref()时，请注意以下几个问题。

- 如果你想获得最佳的执行性能，那么 fetchrow_hashref()就不是最佳的选择。它的执行效率比 fetchrow_array()和 fetchrow_arrayref()低。
- 在默认的情况下，充当键值的数据列名称将沿用它们在 SELECT 语句里的大小写。在 MySQL 里，数据列名称是不区分字母大小写的，所以不管你使用什么样的大小写组合来写数据列的名称，语句都能正确执行。但 Perl 散列的键名称却区分字母大小写，这就可能导致一些问题。为了避免因字母大小写不匹配而导致的意外，你可以给 fetchrow_hashref()方法传递一个 NAME_lc 或 NAME_uc 属性来强制它把数据列名称的大小写统一，如下所示：

  ```
  $hash_ref = $sth->fetchrow_hashref ("NAME_lc"); # use lowercase names
  $hash_ref = $sth->fetchrow_hashref ("NAME_uc"); # use uppercase names
  ```

- 散列中的元素必须对应着一个独一无二的数据列名称。如果你同时对多个数据表进行关联查找而这些数据表又有重复出现的数据列名称，你就无法访问所有数据列值。比如说，如果你发出了下面这样的查询，fetchrow_hashref()就将返回一个只包含 name 这一个元素的散列：

  ```
  SELECT a.name, b.name FROM a INNER JOIN b WHERE a.name = b.name
  ```

  为了避免这种问题，应该使用别名来保证每个数据列都有一个独一无二的名字。比如说，如果你把刚才那个语句改写为下面的样子，就能让 fetchrow_hashref()返回对包含 name 和 name2 两个元素的散列的引用了：

  ```
  SELECT a.name, b.name AS name2 FROM a INNER JOIN b WHERE a.name = b.name
  ```

2. 确定语句返回的数据行的个数

怎样才能知道 SELECT 语句或类似于 SELECT 的语句到底返回了多少个数据行呢？一种办法是在取回时数据行对它们计数。事实上，这是 DBI 查知 SELECT 语句到底返回了多少个数据行的唯一一种可移植手段。就 MySQL 而言，你可以在调用完 execute()之后用刚才的语句句柄去调用 row()方法。但这个办法不能移植到其他的数据库系统使用，DBI 的官方文档里也明确表示不鼓励使用 row()方法去查知 SELECT 语句到底返回了多少个数据行。而且，即便是在 MySQL 里，如果你设置了 mysql-use-result 属性，那么在你取回全部数据行之前，row()返回的统计数字也不可靠。因此，最保险的办法还是在取回数据行时对它们计数。（有关 mysql-use-result 属性的详细说明请参见网上资源附录 H。）

3. 取回只有一个数据行的查询结果

如果结果集里只有一个数据行，就没必要使用循环来取回。比如说，如果你想编写脚本 count_members.pl 来统计美国历史研究会会员人数，可以使用下面的代码：

```
# issue query
my $sth = $dbh->prepare ("SELECT COUNT(*) FROM member");
$sth->execute ();

# read and display query result
my $count = $sth->fetchrow_array ();
$sth->finish ();
$count = "can't tell" if !defined ($count);
print "$count\n";
```

在上面这段代码里，SELECT 语句将只返回一个数据行，所以没必要使用循环，只需调用一次 fetchrow_array()就够了。同时，因为只选取了一个数据列，所以甚至不必把返回值赋值给一个数组。当你在一个标量上下文里调用 fetchrow_array()方法（即你已预知它只会返回一个值而不是一组值）时，它将只返回中选数据行的某一列；若没有数据行中选，则返回 undef。DBI 没有规定 fetchrow_array()在标量上下文里会返回数据行的哪一列。但这对上面这个语句没有丝毫影响。这段代码只检索出了一个值，不可能产生二义性。

虽然结果集里只有一个数据行，但这段代码还是调用了 finish()方法来释放结果集。（在到达最后一行时，fetchrow_array()将隐含地释放，但只有当你第二次调用它时才会有这样的效果。）

还有一种查询最多也只会返回一个数据行，它包含 LIMIT 1 来限制返回的数据行的个数。这种查询经常被用来查找某个数据列里的最大值或者最小值。比如说，下面的查询将把出生日期距今最近的总统的姓名和出生日期打印出来：

```
my $stmt = "SELECT last_name, first_name, birth FROM president"
         . " WHERE birth = (SELECT MAX(birth) FROM president)";
my $sth = $dbh->prepare ($stmt);
$sth->execute ();

my ($last_name, $first_name, $birth) = $sth->fetchrow_array ();
$sth->finish ();
if (!defined ($last_name))
{
  print "Query returned no result\n";
}
else
{
```

```
  print "Most recently born president: $first_name $last_name ($birth)\n";
}
```

使用了 MAX()或 MIN()函数的语句也将只返回一个数据行，所以也不需要使用循环。对于这几种只返回一个数据行的情况，有更加简便的方法，即通过数据库句柄来调用 selectrow_array()方法，它把 prepare()、execute()、数据行取回操作等多项工作合并到一个调用步骤里了。如果执行成功，这个方法将返回一个数组（而不是引用）；如果语句没有返回数据行或者在执行过程中出现了错误，它将返回一个空数组。上面那段代码可以用 selectrow_array()方法改写为如下所示：

```
my $stmt = "SELECT last_name, first_name, birth FROM president"
         . " WHERE birth = (SELECT MAX(birth) FROM president)";
my ($last_name, $first_name, $birth) = $dbh->selectrow_array ($stmt);
if (!defined ($last_name))
{
  print "Query returned no result\n";
}
else
{
  print "Most recently born president: $first_name $last_name ($birth)\n";
}
```

4. 对结果集进行整体处理

如果使用循环来取回数据行，那么，只能按循环返回的顺序来处理数据行，DBI 没有提供任何其他顺序。而且，如果你没有采取适当的维护措施，在你取回下一个数据行之后，上一个数据行就会丢失。这些行为并不总是我们所希望的，比如说，当需要对数据行多次遍历以完成某项统计工作（比如说，第一遍是对数据的粗略分析，而第二遍才是正式的精确统计）时。

把结果集当做整体来访问的方法有几种。首先，你可以使用普通的循环，每取回一个数据行，就把它立刻保存起来。其次，可以使用某个方法一次性地返回整个结果集。不管采取哪种策略，最终得到的都将是一个这样的矩阵：它的每一行分别对应着结果集里的一个数据行，而它的每一列则分别对应着你在 SQL 语句里给出的一个数据列。只要构造出了这个矩阵，想按什么次序来处理其中的元素、想处理多少次就都不是问题了。下面，我们将分别讨论这两种策略。

先说第一种策略——用循环来构造结果集矩阵。其具体思路是：每用 fetchrow_array()方法取回一个数据行，就把该数据行的引用保存到一个数组里。下面这段代码的执行效果与 dump_members.pl 脚本里的“取回+打印”循环差不多，但这里的做法是先把全体数据行保存到一个矩阵里，然后再打印矩阵。这段代码演示了如何确定矩阵有多少个行和列，如何访问矩阵里的各个元素。

```
my @matrix = (); # array of array references

while (my @ary = $sth->fetchrow_array ()) # fetch each row
{
  push (@matrix, [ @ary ]); # save reference to just-fetched row
}

# determine dimensions of matrix
my $rows = scalar (@matrix);
my $cols = ($rows == 0 ? 0 : scalar (@{$matrix[0]}));

for (my $i = 0; $i < $rows; $i++)      # print each row
{
```

```
  my $delim = "";
  for (my $j = 0; $j < $cols; $j++)
  {
    $matrix[$i][$j] = "" if !defined ($matrix[$i][$j]); # NULL value?
    print $delim, $matrix[$i][$j];
    $delim = ",";
  }
  print "\n";
}
```

在确定矩阵行列数时，必须先把它的行数确定下来，因为确定矩阵的列数首先要看它是否为空。如果$row 等于 0，即矩阵是空的，那$col 也将等于 0。因为矩阵的列数就等于它第一行里的元素个数，所以，利用语法@{$matrix[0]}访问整行就能把列数确定下来。

上面的示例把每一个数据行取回为一个数组并保存对它的引用。你也许认为像下面这样用 fetchrow_arrayref()方法来直接检索数据行引用的做法更有效率：

```
my @matrix = (); # array of array references

while (my $ary_ref = $sth->fetchrow_arrayref ())
{
  push (@matrix, $ary_ref); # save reference to just-fetched row
}
```

只可惜这种做法是行不通的，因为 fetchrow_arrayref()会反复使用引用指向的数组。结果矩阵是引用组成的数组，引用全都指向同一个数据行——结果集里的最后一个数据行。因此，如果你想用一次取回一个数据行的办法来构造矩阵，就一定要使用 fetchrow_array()方法，不要使用 fetchrow_arrayref()方法。

第二种策略是调用某个能取回整个结果集来的 DBI 方法。比如说，fetchall_arrayref()方法将返回一个引用，它指向一个元素为引用的数组，每一个数组元素分别指向结果集里某个数据行的内容。虽然听起来很复杂，但从实际效果上讲，这个方法的返回值恰好就是矩阵的引用。fetchall_arrayref()的具体用法是：先依次调用 prepare()和 execute()，再用下面的代码检索结果：

```
# fetch all rows as a reference to an array of references
my $matrix_ref = $sth->fetchall_arrayref ();
```

下面这段代码将确定结果集矩阵的大小并访问其中的元素：

```
# determine dimensions of matrix
my $rows = (!defined ($matrix_ref) ? 0 : scalar (@{$matrix_ref}));
my $cols = ($rows == 0 ? 0 : scalar (@{$matrix_ref->[0]}));

for (my $i = 0; $i < $rows; $i++)      # print each row
{
  my $delim = "";
  for (my $j = 0; $j < $cols; $j++)
  {
    $matrix_ref->[$i][$j] = "" if !defined ($matrix_ref->[$i][$j]); # NULL?
    print $delim, $matrix_ref->[$i][$j];
    $delim = ",";
  }
  print "\n";
}
```

若结果集为空，fetchall_arrayref()将返回一个指向空数组的引用；若执行出错，则返回 undef。因此，如果不启用 RaiseError 属性，在使用它之前，必须先检查返回值。

在确定结果集矩阵的行列数之前，一定要先检查它是否为空。如果你想把矩阵的某一行（如$i）当做一个数组来访问，就需要使用语法@{$matrix_ref->[$i]}。

很明显，利用 fetchall_arrayref()来检索结果集要比使用循环的做法更简单易行，只是用来访问矩阵元素的语法有点古怪。与 fetchall_arrayref()类似、却完成更多事的是 selectall_arrayref()方法，它把 prepare()、execute()、数据行取回循环等一整套流程合并到一个步骤里了。如果打算使用 selectall_arrayref()方法，你只需把语句直接通过数据库句柄传递给它就行了，如下所示：

```
# fetch all rows as a reference to an array of references
my $matrix_ref = $dbh->selectall_arrayref ($stmt);

# determine dimensions of matrix
my $rows = (!defined ($matrix_ref) ? 0 : scalar (@{$matrix_ref}));
my $cols = ($rows == 0 ? 0 : scalar (@{$matrix_ref->[0]}));

for (my $i = 0; $i < $rows; $i++)      # print each row
{
  my $delim = "";
  for (my $j = 0; $j < $cols; $j++)
  {
    $matrix_ref->[$i][$j] = "" if !defined ($matrix_ref->[$i][$j]); # NULL?
    print $delim, $matrix_ref->[$i][$j];
    $delim = ",";
  }
  print "\n";
}
```

5. NULL值的检测

在从数据库检索信息时，经常需要把数据列里的 NULL 值与数值 0 或空字符串区分开来。因为 DBI 会把 NULL 值返回为 undef，所以这做起来并不困难，但你必须按正确的顺序测试。下面这段代码的 3 条 print 语句打印出来的全都是“false!”：

```
$col_val = undef; if (!$col_val) { print "false!\n"; }
$col_val = 0;     if (!$col_val) { print "false!\n"; }
$col_val = "";    if (!$col_val) { print "false!\n"; }
```

这表明以上测试无法区分 undef、数值 0 和空字符串。下面这段代码将打印出两个“false!”，表明测试无法区分 undef 和空字符串：

```
$col_val = undef; if ($col_val eq "") { print "false!\n"; }
$col_val = "";    if ($col_val eq "") { print "false!\n"; }
```

下面这段代码的输出结果还是两个“false!”，说明第二个测试不能把数值 0 和空字符串区分开来。

```
$col_val = "";
if ($col_val eq "") { print "false!\n"; }
if ($col_val == 0)  { print "false!\n"; }
```

要想区分 undef（NULL 值）和非 undef，就必须使用 define()方法。只有先判定了值不是 NULL

之后，才能通过进一步的测试把它与其他类型的“空”值区分开来。比如说：

```
if (!defined ($col_val)) { print "NULL\n"; }
elsif ($col_val eq "")   { print "empty string\n"; }
elsif ($col_val == 0)    { print "zero\n"; }
else                     { print "other\n"; }
```

这些测试的顺序非常重要，如果$col_val 是一个空字符串，第二和第三个比较操作就都将为真。如果这两个测试的顺序颠倒了，你就不能正确地把空字符串和数值 0 区分开来。

8.2.6 在语句字符串引用特殊字符

到目前为止，我们构造出来的语句全都是一些简单地加上了引号的字符串。假如放在引号内的字符串里有带引号的值，这在 Perl 词法上会引起混乱。如果试图插入或者选取一些包含引号、反斜线或二进制数据的值，这在 SQL 层面上也会遇到麻烦。在把一条语句构造为一个 Perl 语言里带引号的字符串时，你必须对语句字符串里每一个引号字符进行转义。如下所示：

```
$stmt = 'INSERT INTO absence VALUES(14,\'2008-09-16\')';
$stmt = "INSERT INTO absence VALUES(14,\"2008-09-16\")";
```

Perl 和 MySQL 里的字符串都可以放在单引号里，也可以放在双引号里。利用这一特点，你往往可以通过混用单、双引号的办法来避免对字符进行转义：

```
$stmt = 'INSERT INTO absence VALUES(14,"2008-09-16")';
$stmt = "INSERT INTO absence VALUES(14,'2008-09-16')";
```

请注意，在使用引号时必须保证字符串将按照你预期的方式得到解释。请考虑下面这些因素。

- 这两种引号在 Perl 语言里是不等效的——只有双引号里的变量名才能被解释为变量。因此，如果你想构造在语句字符串里使用变量的语句，就不能用单引号。比如说，下面两个字符串就不是等价的（假设变量$var 的值是 14）：

  ```
  "SELECT * FROM member WHERE member_id = $var"
  'SELECT * FROM member WHERE member_id = $var'
  ```

  下面是 Perl 对结字符串的解释：

  ```
  "SELECT * FROM member WHERE member_id = 14"
  'SELECT * FROM member WHERE member_id = $var'
  ```

  显然，你应该把第一个字符串传递到 MySQL 服务器去。至于第二个字符串，服务器将会把其中的$var 解释为 member 数据表里的某个数据列的名字。

- 在 MySQL 里，单引号和双引号的含义并不总是相同的。如果服务器运行在禁用了 ANSI_QUOTES SQL 模式的情况下，使用这两种引号当中的任何一种来括住字符串都是可以的。但如果已经启用了 ANSI_QUOTES SQL 模式，就必须使用单引号；双引号只能用于标识符，例如数据库或数据表的名字等。这么说来，用单引号来括住字符串是最保险的办法，因为它在 ANSI_QUOTES 被启用或禁用的情况下都可以使用。

在 Perl 级别上，除把字符串放在双引号里外，还可以使用 qq{}结构，Perl 将把“qq{”和“}”之间的东西视为放在双引号里的字符串。比如说，下面两行代码是等价的：

```
$date = "2008-09-16";
$date = qq{2008-09-16};
```

因为可以在语句里随意使用引号（单引号或双引号）而不必对它们进行转义，所以只要在构造语句时使用了 qq{}，就用不着考虑引号问题了。此外，变量引用也能被正确解释。qq{}的这两个属性在下面的 INSERT 语句里得到了充分的体现：

```
$id = 14;
$date = "2008-09-16";
$stmt = qq{INSERT INTO absence VALUES($id,'$date')};
```

qq 结构的分隔符并非只能使用“{”和“}”，你完全可以写成“qq()”或“qq//”，只要“()”和“\\”不会出现在字符串里。我个人比较喜欢使用“qq{}”，因为字符“}”在语句字符串里出现的几率比“)”和“\”小得多，不容易被误认为是语句的结束字符。就拿“)”来说吧，它在每一条 INSERT 语句里几乎都会出现，所以用“qq()”来引用语句字符串就很容易引起问题。

qq{}结构允许跨越多个代码行。这使我们得以把语句字符串在 Perl 外围代码里凸现出来，如下所示：

```
$id = 14;
$date = "2008-09-16";
$stmt = qq{
  INSERT INTO absence VALUES($id,'$date')
};
```

我们还可以利用这一特点把语句写在多个代码行上以增加可阅读性。比如说，我们在 dump_members.pl 里使用了一条 SELECT 语句：

```
$sth = $dbh->prepare ("SELECT last_name, first_name, suffix, email,"
     . " street, city, state, zip, phone FROM member ORDER BY last_name");
```

利用 qq{}，我们可以把它改写为下面这个样子：

```
$sth = $dbh->prepare (qq{
          SELECT
            last_name, first_name, suffix, email,
            street, city, state, zip, phone
          FROM member
          ORDER BY last_name
        });
```

虽说放在双引号里的字符串也允许跨越多行，但我认为“qq{”和“}”要比两个孤零零的“"”字符更抢眼，能使语句更容易阅读。这两种格式在本书里都有使用，至于哪一种更好，就自己判断吧。

qq{}结构解决了 Perl 词法层面上的引号问题，它使我们能够在字符串里随意使用引号而不会引起 Perl 的抱怨。但我们必须更进一步地考虑到 SQL 层面上的语法问题。下面这段代码准备把一条新数据行插入到 member 数据表里：

```
$last = "O'Malley";
$first = "Brian";
$expiration = "2013-09-01";
$rows = $dbh->do (qq{
  INSERT INTO member (last_name,first_name,expiration)
  VALUES('$last','$first','$expiration')
});
```

do()方法发送给 MySQL 的结果字符串将是下面这个样子：

```
INSERT INTO member (last_name,first_name,expiration)
VALUES('O'Malley','Brian','2013-09-01')
```

这是个不合法的 SQL 语句，因为单引号里的字符串里又出现了单引号字符（'O'Malley'）。我们在第 7 章里曾遇到过类似的问题，当时是用 mysql_real_escape_string()函数来解决的。DBI 也提供了一个类似的机制：当需要在语句里使用带引号的值时，可以调用 quote()方法，然后把这个方法的返回值用在语句里。上面那个例子写成下面这样更好：

```
$last = $dbh->quote ("O'Malley");
$first = $dbh->quote ("Brian");
$expiration = $dbh->quote ("2013-09-01");
$rows = $dbh->do (qq{
  INSERT INTO member (last_name,first_name,expiration)
  VALUES($last,$first,$expiration)
});
```

现在，do()方法发送给 MySQL 的字符串将是下面这样，带引号的字符串里的引号字符都得到了正确的转义处理：

```
INSERT INTO member (last_name,first_name,expiration)
VALUES('O\'Malley','Brian','2013-09-01')
```

注意，语句字符串里的$last 和$first 都不应该再用引号括起来，因为 quote()方法会替你完成。如果你还画蛇添足地加上了引号，语句里的引号就太多了，就像下面这个例子这样：

```
$value = "paul";
$quoted_value = $dbh->quote ($value);

print "The quoted value is: $quoted_value\n";
print "The quoted value is: '$quoted_value'\n";
```

这些语句将产生如下所示的输出：

```
The quoted value is: 'paul'
The quoted value is: ''paul''
```

很明显，第二个字符串里的引号太多了。

8.2.7 占位符与预处理语句

在前面几节里，我们在构造语句时都是把将被插入或者将被用在选择条件里的数据值直接放到语句字符串里的。这不是必需的。DBI 允许在语句字符串里使用特殊的符号，即占位符，然后在执行这个语句时再用具体的值替换掉占位符，这叫做“将值绑定到语句”。这种做法有两个原因：你不需要明确地调用 quote()方法就能获得同样的字符引用效果；如果语句会在一个循环里反复执行很多次，你可以先预处理再多次执行，从而避免每次执行前增加预处理开销，这将大幅改善脚本的执行性能。

为了说明占位符的作用和使用方法，假设现在正是一个新学期的开始，你需要把考试记分项目中的 student 数据表里的旧记录清理干净，并用保存在某个文件里的新学生名单来重新建立这个数据表。如果不使用占位符，你可以像下面这样删除数据表里的旧数据和加载新名字：

```
$dbh->do (qq{ DELETE FROM student } );  # delete existing rows
while (<>)                              # read each input line,
{                                       # use it to add a new row
  chomp;
```

```
    $_ = $dbh->quote ($_);
    $dbh->do (qq{ INSERT INTO student SET name = $_ });
  }
```

这段代码需要由你本人去调用 quote()方法来处理数据值里的特殊字符。它的执行效率太低，因为 INSERT 语句的基本格式每次都一样，do()方法必须在每次循环时 prepare()和 execute()各做一次调用。要是能在进入循环之前，只在构造 INSERT 语句时调用一次 prepare()，在进入循环之后，每次循环只需调用 execute()而不必再调用 prepare()，把对 prepare()方法的调用次数减少为一次，那可就太理想了。利用 DBI 就能实现完成这些任务：

```
$dbh->do (qq{ DELETE FROM student } );  # delete existing rows
my $sth = $dbh->prepare (qq{ INSERT INTO student SET name = ? });
while (<>)                              # read each input line,
{                                       # use it to add a new row
  chomp;
  $sth->execute ($_);
}
```

一般说来，如果你发现需要在一个循环里多次调用 do()方法，就应该把 prepare()调用安排在进入循环之前，在循环里则只调用 execute()方法。注意到 INSERT 语句里的问号了吗？它就是占位符。调用 execute()方法时，语句发送给服务器前占位符将被替换为一个具体的值。DBI 将自动给值里的特殊字符加上引号，用不着调用 quote()。

在使用占位符时要注意以下几项。

- 不要给语句字符串里的占位符加上引号，否则，它将不会被识别为占位符。
- 不要使用 quote()方法来指定占位符的值，否则，这个值将有多余的引号。
- 在同一个语句字符串里允许出现一个以上的占位符，但这么做就必须保证传递给 execute()函数的值的个数和占位符的个数是一样的。
- 每个占位符只能对应于一个值。比如说，如果你要指定多个数据值，就不能像下面这样去构造和执行语句：

  ```
  my $sth = $dbh->prepare (qq{
    INSERT INTO member last_name, first_name VALUES(?)
  });
  $sth->execute ("Adams,Bill,2014-07-19");
  ```

  你必须分别指定值，并为每个值提供占位符：

  ```
  my $sth = $dbh->prepare (qq{
    INSERT INTO member last_name, first_name VALUES(?,?,?)
  });
  $sth->execute ("Adams","Bill","2014-07-19");
  ```

- 如果需要把某个占位符的值替换为 NULL 值，必须使用 undef。
- 占位符和 quote()方法只适用于数据值，SELECT 之类的关键字或者数据库名、数据表名、数据列名之类的标识符是不允许用占位符来“占位子”的；否则，这些关键字或标识符就会被加上引号，从而导致这条语句因语法错误而执行失败。

对于某些数据库引擎，占位符除了提高循环语句的效率以外，还有另外一个与执行效率有关的好处。一些数据库引擎会把预处理语句和语句执行计划缓存起来。这样，当服务器再次接收到同样的语句时，语句就可以立刻被重用而不必预处理。把查询缓存起来的做法特别有利于复杂的 SELECT 语句。

与通过在语句字符串里直接嵌入特定数据值的办法构造出来的语句相比，利用占位符构造出来的语句更具通用性，因而更适合缓存起来反复使用。

MySQL 不缓存执行计划。MySQL 有一个查询缓存，但该缓存只用来缓存查询字符串的结果集，不缓存执行计划。对查询缓存的讨论见第 5 章。

在默认的情况下，MySQL 也不缓存预处理语句。把参数绑定到占位符的动作发生在客户端的 DBD::mysql 模块里。不过，MySQL C 客户端库里实现的二进制协议既允许语句在服务器端接受预处理，也允许由服务器来处理参数绑定事宜。DBD::mysql 模块可以受益于这种能力。

为了打开服务器端的预处理语句和参数绑定功能，只需要启用 mysql_server_prepare 选项。比如说，给定一个数据库句柄$dbh，下面这条语句就能让你达到这一目的：

```
$dbh->{mysql_server_prepare} = 1;
```

如果想禁用服务器端的预处理语句功能，只要把这个选项设置为零就可以了。

要想获得 mysql_server_prepare 支持，最好是使用 DBD::mysql 3.0009 或更高的版本，这是因为在部分较早的版本里，该选项的默认值有一些变化。

即使不打算使用 MySQL 的服务器端预处理语句功能，利用占位符来编写语句也是有好处的：当把脚本用于某种不缓存执行计划的数据库引擎时，有占位符的语句要比没有占位字符的语句执行效率更高。

神秘的 undef

有些执行语句字符串的 DBI 方法，如 do()和 selectrow_array()，允许你为语句里的任意多个“？”字符提供占位符值。比如说，可以这样修改数据行：

```
my $rows = $dbh->do (
            "UPDATE member SET expiration = ? WHERE member_id = ?",
            undef, "2007-01-01", 14);
```

或者这样提取数据行：

```
my $ref = $dbh->selectrow_arrayref (
           "SELECT * FROM member WHERE member_id = ?",
           undef, 14);
```

但是这两段代码都有一个看起来没有任何意义的神秘的 undef 参数出现在占位符之前。造成这种现象的原因是这样的：在这些允许使用占位符作为参数的语句执行方法里，在真正的参数之前还存在着一个用来设定语句处理属性的参数虽然没有写出。这些属性虽然很少会用到（如果还真有人用的话），但与之对应的参数却不能省略，因而指定为 undef。

8.2.8 把查询结果绑定到脚本变量

占位符允许你直到语句被执行时才把真正要用到的值替换到语句字符串里。换句话说，这相当于允许语句使用“输入参数”。与此相对应，DBI 还提供了一种名为“参数绑定”的输出操作以允许语句使用“输出参数”。这个操作能够在你取回一个数据行时自动把语句数据列的值检索到变量中，不需要你对这些变量赋值。

假设你有一个用来从 member 数据表检索姓名的查询，你可以让 DBI 把被选取数据列的值自动赋给 Perl 变量。当你取回一个数据行时，这些变量将自动更新为相应的数据列值，这就使你的检索操作变得非常有效率。下面这个例子演示了如何把数据列绑定到某个变量，以及如何在用来取回数据行的循环里访问这些变量：

```
my ($last_name, $first_name, $suffix);
my $sth = $dbh->prepare (qq{
            SELECT last_name, first_name, suffix
            FROM member ORDER BY last_name, first_name
          });
$sth->execute ();
$sth->bind_col (1, \$last_name);
$sth->bind_col (2, \$first_name);
$sth->bind_col (3, \$suffix);
print "$last_name, $first_name, $suffix\n" while $sth->fetch ();
```

bind_col()调用应出现在 execute()之后、取回数据行之前，它的参数有两个：数据列的编号，绑定到该数据列的变量的引用。数据列的编号从 1 开始。

除了像上面这样每次使用一系列 bind_col()函数，还有一个办法是把所有的变量引用一次性地全部传递到一个 bind_columns()函数，如下所示：

```
my ($last_name, $first_name, $suffix);
my $sth = $dbh->prepare (qq{
            SELECT last_name, first_name, suffix
            FROM member ORDER BY last_name, first_name
          });
$sth->execute ();
$sth->bind_columns (\$last_name, \$first_name, \$suffix);
print "$last_name, $first_name, $suffix\n" while $sth->fetch ();
```

bind_columns()调用也应该出现在 execute()之后、取回数据行之前。

8.2.9　设定连接参数

建立服务器连接最直接的办法是把所有的连接参数设定为 connect()方法的参数：

```
my $dsn = "DBI:mysql:db_name:host_name";
my $dbh = DBI->connect ($dsn, user_name, password);
```

如果省略了连接参数，DBI 将根据以下规则去确定。

- 如果事先设定了 DBI_DSN 环境变量且数据源名称（即 DSN）没有定义或者是一个空字符串，将使用 DBI_DSN 环境变量的值作为数据源。如果事先设定了 DBI_USER 和 DBI_PASS 环境变量且用户名和口令没有定义（注意，不包括它们是空字符串的情况），就将使用 DBI_USER 和 DBI_PASSWORD 环境变量的值作为用户名和口令。在 Windows 系统上，如果没有定义用户名，就将使用 USER 环境变量的值作为用户名。
- 如果省略了主机名，DBI 将尝试连接本地主机。
- 如果把用户名设定为 undef 或一个空字符串，就默认使用你的 Unix 登录名进行连接。在 Windows 系统上，用户名的默认值是 ODBC。
- 如果把口令设定为 undef 或一个空字符串，就不发送口令。

你可以在 DSN 里设定一些选项，追加在字符串的开头，并且必须用分号彼此隔开。比如说，你

可以用 mysql_read_default_file 选项来指定选项文件路径名：

```
my $dsn = "DBI:mysql:sampdb;mysql_read_default_file=/home/paul/.my.cnf";
```

这将使脚本在执行时到指定文件里去读取连接参数。比如说，如果/u/paul/.my.cnf 文件有着如下所示的内容：

```
[client]
host=localhost
user=sampadm
password=secret
```

connect()调用将尝试以用户名 sampdb 和口令 secret 来连接 localhost 上的 MySQL 服务器。在 Unix 系统上，你还可以通过像下面这样给出一个选项文件名参数来让脚本读取它当前使用者的选项文件：

```
my $dsn = "DBI:mysql:sampdb;mysql_read_default_file=$ENV{HOME}/.my.cnf";
```

$ENV{HOME}给出了这个脚本的当前使用者的主目录的路径名，所以连接参数将来自当前用户自己的选项文件。如果以这种方式编写脚本，就用不着在脚本里写出连接参数了。

mysql_read_default_file 选项只能让脚本去读取指定选项文件里的连接参数，如果还想让它读取全局选项文件（如 Unix 系统上的/etc/my.cnf 文件或者 Windows 系统上的 C:\my.ini 文件）里的连接参数，这个选项就不能胜任了。如果想让脚本读取所有标准选项文件里的连接参数，就应该使用 mysql_read_default_group 选项。这个选项将把[client]和你指定的那个设置段里的连接参数都读取出来。比如说，如果你有一些专供与 sampdb 数据库有关的脚本使用的选项，就可以把它们列在一个[sampdb]设置段里，然后像下面这样使用数据源值：

```
my $dsn = "DBI:mysql:sampdb;mysql_read_default_group=sampdb";
```

如果只想使用标准选项文件里的[client]设置段里的连接参数，就应该这样指定选项：

```
my $dsn = "DBI:mysql:sampdb;mysql_read_default_group=client";
```

关于用于指定数据源字符串的选项的详细介绍请参见在线资源附录 H，关于 MySQL 选项文件格式的详细介绍请参见附录 F。

但是，在 Windows 系统上使用 mysql_read_default_file 选项会遇到这样一个难题：文件路径名通常以一个驱动盘符和一个冒号开始，可 DBI 却会把这个冒号解释为 DSN 字符串里的分隔符。有个能避开这一限制的办法，但这个办法有点笨拙。

(1) 利用 chdir()把路径切换到选项文件所在驱动的根目录，这样不带驱动盘符的路径名就将被解释为是相对于驱动的。

(2) 把 DSN 里的 mysql_read_default_file 选项的值设置为文件名，但不要写出驱动盘符或分号。

(3) 如果需要在连接上 MySQL 服务器之后再回到当前子目录里来，就必须在调用 connect()之前先把当前目录的路径名保存起来，等连接操作完成后，再通过 chdir()调用重新返回。

下面这段代码演示了怎样才能使用选项文件 C:\my.ini。（请注意：在 Perl 字符串里，Windows 路径名里的反斜线符“\”必须写为斜线符“/”。）

```
# save current directory pathname
use Cwd;
```

8

```
my $orig_dir = cwd ();
# change to root dir of drive where file is located
chdir ("C:/") or die "Cannot chdir: $!\n";
# connect using parameters in C:\my.ini
my $dsn = "DBI:mysql:sampdb:localhost;mysql_read_default_file=/my.ini";
my %conn_attrs = (RaiseError => 1, PrintError => 0, AutoCommit => 1);
my $dbh = DBI->connect ($dsn, undef, undef, \%conn_attrs);
# change back to original directory
chdir ($orig_dir) or die "Cannot chdir: $!\n";
```

使用选项文件并不妨碍你在 connect()调用里给出连接参数（比如说，如果你想让脚本作为某个特定的用户去建立连接的话）。在 connect()调用里明确给出的连接参数（主机名、用户名、关键字）将覆盖来自选项文件的连接参数。比如说，如果你想让脚本去分析来自命令行的--host 和--user 选项并使用它们给出的值，只要在命令行上把它们写出来，就可以让它们优先于选项文件里的相同选项。这么做很有道理，因为这也正是 MySQL 自带客户程序的“标准”行为，而你的 DBI 脚本应该与这种行为保持一致。

至于在本章开发的其他一些命令行脚本，我将使用一些标准的连接设置并省略对有关代码的分析。这个技巧只在下面演示一次，其他地方的讨论将集中在要编写的每个脚本的主体上：

```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;

# parse connection parameters from command line if given

use Getopt::Long;
$Getopt::Long::ignorecase = 0; # options are case sensitive
$Getopt::Long::bundling = 1;   # -uname = -u name, not -u -n -a -m -e

# default parameters - all undefined initially
my ($host_name, $password, $port_num, $socket_name, $user_name);

GetOptions (
  # =i means an integer value is required after option
  # =s means a string value is required after option
  "host|h=s"      => \$host_name,
  "password|p=s"  => \$password,
  "port|P=i"      => \$port_num,
  "socket|S=s"    => \$socket_name,
  "user|u=s"      => \$user_name
) or exit (1);

# construct data source
my $dsn = "DBI:mysql:sampdb";
$dsn .= ";host=$host_name" if $host_name;
$dsn .= ";port=$port_num" if $port_num;
$dsn .= ";mysql_socket=$socket_name" if $socket_name;
$dsn .= ";mysql_read_default_group=client";

# connect to server
```

```
my %conn_attrs = (RaiseError => 1, PrintError => 0, AutoCommit => 1);
my $dbh = DBI->connect ($dsn, $user_name, $password, \%conn_attrs);
```

这段代码对 DBI 进行了初始化。它先对命令行上给出的连接参数进行了处理，然后使用来自命令行或者它在标准选项文件的[client]设置段里找到的连接参数去连接 MySQL 服务器。如果你把连接参数安排在你的选项文件里，就用不着在运行使用这代码的脚本时输入它们了。

每个脚本的结尾部分也都是同样的代码，它们将断开连接并退出执行，如下所示：

```
$dbh->disconnect ();
```

8.4 节讲到 Web 编程技术时，我们将对连接代码稍做修改，但基本思路仍将是相同的。

MySQL 的标准客户程序与 Getopt 模块在处理命令行参数方面有一个不幸的区别。MySQL 的标准客户程序的选项处理代码很完备，口令值是否紧跟在口令选项（--password 或--p）的后面不会导致任何歧义。如果用户没有在口令选项的后面立刻给出口令值，它将提示用户输入。

再看 Getopt 模块，它在一定程度上也允许紧随在--password 和--p 选项后面的口令值是可选的，但只有以下两种情况不会导致歧义：其一是把该选项放在整个命令行的最后，其二是在该选项的后面紧跟着另一个选项。如果某个脚本使用了 Getopt 模块，并且在运行时还需要一个数据表的名字作为参数，像下面这样执行这个脚本时，Getopt 模块将把 mytbl 错误地解释为口令值而不再提示你输入口令：

```
% ./myscript.pl -u paul -p mytbl
```

为避免这类问题，前面 Perl 框架里的代码要求口令选项在给出时必须带有一个值。

8.2.10 调试

要调试不能正常工作的 DBI 脚本，有两种比较常用的方法，它们既可以单独使用，也可以联合作战。第一种办法是在脚本里安排一些打印语句，这样你就可以随意安排调试工作的输出信息，但你必须手动添加语句。第二种办法是使用 DBI 内建的跟踪功能，它的好处是更普适和更系统，在启用后自动完成。DBI 跟踪功能还能让你看到驱动的操作信息，这是其他调试手段做不到的。

8

1. 利用print语句调试

有一个常见的问题：“我的语句在 mysql 客户程序里执行得挺好的，可在 DBI 脚本里就不行了。为什么会这样？”这类问题最为常见的起因是：DBI 脚本发送的语句并不是你以为的那个。如果在执行语句前先打印出来，你会惊讶于实际发送到服务器的东西。假设你在 mysql 客户程序里敲入了下面这个语句：

```
mysql> INSERT INTO member (last_name,first_name,expiration)
    -> VALUES('Brown','Marcia','2012-06-03');
```

接着，你在一个 DBI 脚本里试图去做同样的事情（记得要去掉末尾的分号）：

```
$last = "Brown";
$first = "Marcia";
$expiration = "2012-06-03";
$stmt = qq{
  INSERT INTO member (last_name,first_name,expiration)
  VALUES($last,$first,$expiration)
};
$rows = $dbh->do ($stmt);
```

查询还是那个查询，但它这一次却没有正确执行。查询真的还是那个查询吗？把它打印出来看看：

```
print "$stmt\n";
```

结果如下所示：

```
INSERT INTO member (last_name,first_name,expiration)
VALUES(Brown,Marcia,2012-06-03)
```

根据这个输出结果，可以清楚地看出前后两条语句根本就不一样：VALUES()列表里的值没有引号。这个漏洞有两种办法可以弥补。其一，使用 quote()方法，如下所示：

```
$last = $dbh->quote ("Brown");
$first = $dbh->quote ("Marcia");
$expiration = $dbh->quote ("2012-06-03");
$stmt = qq{
  INSERT INTO member (last_name,first_name,expiration)
  VALUES($last,$first,$expiration)
};
$rows = $dbh->do ($stmt);
```

其二，使用占位符来指定语句，然后再把将要插入的语句值传递给 do()方法作为其参数，如下所示：

```
$last = "Brown";
$first = "Marcia";
$expiration = "2012-06-03";
$stmt = qq{
  INSERT INTO member (last_name,first_name,expiration)
  VALUES(?,?,?)
};
$rows = $dbh->do ($stmt, undef, $last, $first, $expiration);
```

不过，如果使用的是第二种办法，就无法利用打印语句去查看语句的完整内容了，因为用值来替换占位符的动作只有在执行 do()时才会发生。如果使用了占位符，DBI 内建的跟踪调试功能往往会更有帮助。

2. 利用跟踪机制调试

DBI 提供了一个能够生成调试信息的跟踪机制，能找出脚本行为异常的原因。该机制从 0（关闭）到 15（信息量最大）共有 16 个级别。一般说来，第 1 级到第 4 级最有用。第 2 级能让你看到你正在执行的语句的文本（包括已完成占位符替换后的结果）、quote()方法的调用结果等，这些信息对跟踪问题有着巨大的帮助。

你可以对各个脚本进行跟踪调试（办法是调用 trace()方法），也可以对你运行的全部 DBI 脚本进行跟踪调试（办法是设置 DBI_TRACE 环境变量）。

在调用 trace()方法时，应提供一个跟踪级别参数和一个可选的文件名。如果没有给出文件名，调试信息将被发送到 STDERR；如果给出了这个文件名，调试信息就被发送到指定的文件里。下面这个调用启动了第 1 级，调试信息将送往 STDERR：

```
DBI->trace (1);
```

下面这个启动了第 2 级，调试信息将送往 trace.out 文件：

```
DBI->trace (2, "trace.out");
```

如果想关闭跟踪，请把跟踪级别设定为 0：

```
DBI->trace (0);
```

发出 DBI->trace()调用将跟踪所有的 DBI 操作。更细致的办法是只在特定的句柄级别启用跟踪功能，这在你已经对脚本的问题所在有了比较准确的推断、并且不想浪费时间去看长篇跟踪报告的时候会非常有用。比如说，如果在某个特定的 SELECT 查询上遇到了问题，可以只跟踪与该查询相关联的语句句柄：

```
$sth = $dbh->prepare (qq{ SELECT ... }); # create the statement handle
$sth->trace (1);                         # enable tracing on the statement
$sth->execute ();
```

如果你在 trace()调用里给出了一个文件名，那么所有的跟踪输出（不管它们是来自 DBI 全局还是来自某个特定的句柄）都将写入这个文件。

除调用 trace()外，还可以使用 TraceLevel 属性，它允许设置或者读取某给定句柄上的跟踪级别，如下所示：

```
$dbh->{TraceLevel} = 3;               # set database handle trace level
my $cur_level = $sth->{TraceLevel};   # get statement handle trace level
```

如果想启用全局跟踪，使它能够作用于你执行的全部脚本，可以在 shell 里设置 DBI_Trace 环境变量，语法取决于所使用的 shell：

- 如果使用的是 csh 或 tcsh，

```
% setenv DBI_TRACE value
```

- 如果使用的是 sh、ksh 或 bash：

```
$ export DBI_TRACE=value
```

- 如果使用的是 Windows 系统：

```
C:\> set DBI_TRACE=value
```

value 的格式在各种 shell 里都一样：如果它是一个整数 *n*，则表示跟踪级别是 *n*，输出将被写到 STDERR；如果它是一个文件名，则表示跟踪级别是 2，输出将被写到指定文件；如果它是 *n=file_name*，则表示跟踪级别是 *n*，输出将被送往指定文件。下面是几个使用 tcsh 语法的例子：

- 跟踪级别是 1，输出将被写到 STDERR：

```
% setenv DBI_TRACE 1
```

- 跟踪级别是 1，输出将被写到 trace.out 文件：

```
% setenv DBI_TRACE 1=trace.out
```

- 跟踪级别是 2，输出将被写到 trace.out 文件：

```
% setenv DBI_TRACE trace.out
```

使用 DBI_Trace 环境变量的好处是你不用修改脚本就能启用跟踪调试机制。但如果你从 shell 里启用了对某个文件的跟踪，千万要记得在解决问题之后把它关闭掉。要知道，调试输出一直追加在跟踪文件的末尾而不会覆盖它，如果你自己不注意，这个文件就可能变得非常大。因此，在某个 shell 启动文件（如.cshrc、.tcshrc、.login 或.profile 等）里定义 DBI_Trace 环境变量是极不可取的！

下面这些命令用来关闭各种命令解释器环境中的 DBI_TRACE 功能：

- 在 csh 或 tcsh 里，

```
% setenv DBI_TRACE 0
```

```
% unsetenv DBI_TRACE
```

- 在 sh、ksh 或 bash 里，

```
$ unset DBI_TRACE
$ export DBI_TRACE=0
```

- 在 Windows 系统上，

```
C:\> unset DBI_TRACE
C:\> set DBI_TRACE=0
```

8.2.11　使用结果集的元数据

你可以使用 DBI 来访问结果集的元数据，它们是一些关于查询命令所选取的数据行的描述性信息。要取得元数据，就要访问与生成结果集的查询相关联的语句句柄的属性。它们有的是所有数据库驱动都具备的 DBI 标准属性(如 NUM_OF_FIELDS,结果集里的数据列个数)；另外一些则是 DBD::mysql (用于 DBI 的 MySQL 驱动) 为 MySQL 提供的专用属性，这类专用属性 (如 mysql_max_length 属性，各数据列的数据值最大宽度）往往不适用于其他的数据库系统。如果你在脚本里使用了 MySQL 的专用属性，你的脚本就可能无法移植到其他数据库里使用；但从另一方面讲，它们的确能让你更容易地获得你想要的信息。

必须掌握好访问元数据的时机。在你完成对 prepare()和 execute()调用之前，SELECT 语句的结果集属性是不可用的；可当你使用某个数据行取回函数到达结果集末尾，或者调用了 finish()方法之后，这些属性可能已经失效了。

下面的示例演示了 DBI 元数据通用属性 NUM_OF_FIELDS（结果集里的数据列个数）和 NAME（结果集里各数据列的名称）以及 MySQL 元数据的专用属性 mysql_max_length 的用法。结合这几个属性提供的信息写脚本 box_out.pl，它从 SELECT 查询生成的输出报告将与交互式客户程序 mysql 生成的输出报告有同样的表格型格式。以下代码就是 box_out.pl 脚本的主要部分。你可以把 SELECT 语句替换为任何其他的语句，输出例程与具体的语句无关：

```
my $sth = $dbh->prepare (qq{
            SELECT last_name, first_name, suffix, city, state
            FROM president ORDER BY last_name, first_name
          });
$sth->execute (); # attributes should be available after this call

# actual maximum widths of column values in result set
my @wid = @{$sth->{mysql_max_length}};
# number of columns in result set
my $ncols = $sth->{NUM_OF_FIELDS};

# adjust column widths if data values are narrower than column headings
# or than the word "NULL"
for (my $i = 0; $i < $ncols; $i++)
{
  my $name_wid = length ($sth->{NAME}->[$i]);
  $wid[$i] = $name_wid if $wid[$i] < $name_wid;
  $wid[$i] = 4 if $wid[$i] < 4;
}
```

```
# print tabular-format output
print_dashes (\@wid, $ncols);             # row of dashes
print_row ($sth->{NAME}, \@wid, $ncols);  #column headings
print_dashes (\@wid, $ncols);             #row of dashes
while (my $ary_ref = $sth->fetchrow_arrayref ())
{
  print_row ($ary_ref, \@wid, $ncols);    #row data values
}
print_dashes (\@wid, $ncols);             #row of dashes
```

在调用 execute()方法初始化语句之后，我们立刻访问了我们需要的元数据。$sth->{NUM_OF_FIELDS}是一个标量变量，它能告诉我们结果集里有多少个数据列。$sth->{NAME}和$sth->{mysql_max_length}则分别给出了数据列的名字和它们的数据值最大宽度。这两个属性的值都是一个数组引用，数组里的各个元素分别对应着结果集里的各个数据列，它们的排列顺序与数据列在语句里出现的顺序相同。

其他计算与第 7 章里编写的 exec_stmt 程序中的计算大同小异。比如说，为避免出现输出不整齐，如果某数据列名字的长度大于其数据值的长度，与之对应的输出列的宽度就将相应地调整为该数据列名字的长度。

下面是输出函数 print_dashes()和 print_row()的代码。它们也与 exec_stmt 里的有关代码大同小异：

```
sub print_dashes
{
my $wid_ary_ref = shift;  # reference to array of column widths
my $cols = shift;         # number of columns

  for (my $i = 0; $i < $cols; $i++)
  {
    print "+", "-" x ($wid_ary_ref->[$i]+2);
  }
  print "+\n";
}

# print row of data.  (doesn't right-align numeric columns)

sub print_row
{
my $val_ary_ref = shift;  # reference to array of column values
my $wid_ary_ref = shift;  # reference to array of column widths
my $cols = shift;         # number of columns

  for (my $i = 0; $i < $cols; $i++)
  {
    printf "| %-*s ", $wid_ary_ref->[$i],
           defined ($val_ary_ref->[$i]) ? $val_ary_ref->[$i] : "NULL";
  }
  print "|\n";
}
```

下面是 tabular.pl 脚本生成的输出：

```
+------------+---------------+--------+---------------------+-------+
| last_name  | first_name    | suffix | city                | state |
+------------+---------------+--------+---------------------+-------+
| Adams      | John          | NULL   | Braintree           | MA    |
| Adams      | John Quincy   | NULL   | Braintree           | MA    |
| Arthur     | Chester A.    | NULL   | Fairfield           | VT    |
| Buchanan   | James         | NULL   | Mercersburg         | PA    |
| Bush       | George H.W.   | NULL   | Milton              | MA    |
| Bush       | George W.     | NULL   | New Haven           | CT    |
| Carter     | James E.      | Jr.    | Plains              | GA    |
...
```

我们的下一个脚本将利用数据列的元数据生成另外一种格式的输出。这个名为 show_member.pl 的脚本能让你快速查看美国历史研究会会员的记录而不必输入任何语句。给定某位会员的姓氏，这个脚本将把中选项显示为如下所示的格式：

```
% ./show_member.pl artel
last_name:  Artel
first_name: Mike
suffix:
expiration: 2011-04-16
email:      mike_artel@venus.org
street:     4264 Lovering Rd.
city:       Miami
state:      FL
zip:        12777
phone:      075-961-0712
interests:  Civil Rights,Education,Revolutionary War
member_id:  63
```

show_member.pl 脚本还接受会员编号或 SQL 模式匹配模板（用来匹配多个姓氏）作为其参数。下面的第一条命令将检索出 23 号会员的记录，第二条命令将检索出那些姓氏以字母“C”开头的会员的记录：

```
% ./show_member.pl 23
% ./show_member.pl C%
```

下面是 show_member.pl 脚本的主要正文。各输出行的标签是它利用 NAME 属性而确定的，结果集里的数据列个数则是它利用 NUM_OF_FIELDS 属性而确定的：

```
my $count = 0;  # number of entries printed so far
my @label = (); # column label array
my $label_wid = 0;

while (@ARGV)   # run query for each argument on command line
{
  my $arg = shift (@ARGV);

  # default is to do a pattern search by last name...
  my $clause = "last_name LIKE " . $dbh->quote ($arg);
  # ...but do ID search instead if argument is numeric
  $clause = "member_id = " . $dbh->quote ($arg) if $arg =~ /^\d+$/;

  # issue query
```

```
my $sth = $dbh->prepare (qq{
            SELECT * FROM member
            WHERE $clause
            ORDER BY last_name, first_name
          });
$sth->execute ();

# get column names to use for labels and
# determine max column name width for formatting
# (but do this only the first time through the loop)
if ($label_wid == 0)
{
  @label = @{$sth->{NAME}};
  foreach my $label (@label)
  {
    $label_wid = length ($label) if $label_wid < length ($label);
  }
}

# read and display query result
my $matches = 0;
while (my @ary = $sth->fetchrow_array ())
{
  # print newline before 2nd and subsequent entries
  print "\n" if ++$count > 1;
  foreach (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
  {
    # print label
    printf "%-*s", $label_wid+1, $label[$i] . ":";
    # print value, if there is one
    print " ", $ary[$i] if defined ($ary[$i]);
    print "\n";
  }
  ++$matches;
}
print "\nNo match was found for \"$arg\"\n" if $matches == 0;
}
```


不管一个数据行包含有多少个数据，show_member.pl 脚本的目的就是把它的完整内容显示出来。这个脚本先使用 SELECT *检索所有的数据列，然后再利用 NAME 属性把它们的名字查出来。这样，即使 member 数据表在今后增加或者删除了一些数据列，这个脚本也不需要修改。

如果你只想知道数据表里到底有哪些数据列而不打算检索其中的数据行，可以使用一个如下所示的语句：

```
SELECT * FROM tbl_name WHERE FALSE
```

WHERE FALSE 子句对所有数据行来说结果都是 FALSE，所以这条语句的执行效果是在不返回任何数据行的情况下生成数据列的元数据。当你像平时那样调用完 prepare()和 execute()方法之后，就可以从@{$sth->{NAME}}查知各数据列的名字了。但我还得提醒大家一句：这种利用一个“空”查询来查知数据列名字的小技巧虽然适用于 MySQL，却不见得能移植到别的数据库系统里运用。

如果想进一步了解 DBI 和 DBD::mysql 提供的属性，请参阅在线资源附录 H。那么，是避免使用 MySQL 专用属性以保持可移植性呢？还是以牺牲可移植性为代价去使用它们呢？你自己判断吧。

8.2.12　实现事务处理

用 DBI 脚本来实现事务处理机制的办法之一是明确地发出 SET AUTOCOMMIT、START、TRANSACTION、COMMIT 和 ROLLBACK 等语句。（对这几条语句的详细介绍请参见 2.13 节。）DBI 模块提供了一套手段来实现事务处理操作。这套手段由 DBI 方法和属性构成，它们将自动替你发出与事务有关的正确的 SQL 语句。你可以把使用这套手段编写出来的脚本移植到其他也支持事务的数据库系统里去使用，但 SQL 语句除外。

要想使用 DBI 事务处理机制，必须满足以下几个条件。

- 你的 MySQL 服务器必须至少支持一个事务安全存储引擎，如 InnoDB 或 Falcon。如何查知 MySQL 服务器是否支持它们的办法请参见 2.6.1 节中的第 1 小节。
- 应用程序必须使用事务安全类型的数据表，否则，就必须先用 ALTER TABLE 改变它们的类型。下面这条语句能把给定数据表 tbl_name 的类型修改为 InnoDB：

  ```
  ALTER TABLE tbl_name ENGINE = InnoDB;
  ```

如果以上条件全部得到了满足，就可以在 DBI 脚本里按以下步骤实现事务处理了。

(1) 禁用（或临时挂起）自动提交模式，由你本人来控制在什么时候提交 SQL 语句。

(2) 把构成某个事务的语句放在一个 eval 块执行，并在执行时启用 RaiseError 属性、禁用 PrintError 属性。这样一来，任何错误都将结束这个块，但不打印出错消息。如果这个块执行成功，块内的最后一个操作应该是 invoke commit()，也就是提交本次事务。

(3) 在 eval 语句块执行终止后，立刻检查其终止状态。如果执行出错，就要调用 rollback()方法来撤销事务并给出相应的出错信息（如果需要的话）。

(4) 根据需要恢复自动提交模式和出错处理属性。

接下来的例子将演示如何实现这一思路。故事仍是第 2 章中的一幕，我们当时正在学习如何从 mysql 客户程序以手动方式发出与事务相关联的语句。具体地说，你发现你在 score 数据表里把两名学生的分数弄混了，需要把它们交换过来：给 8 号学生的分数是 18，给 9 号学生的分数是 13，而这两个分数应该调换过来。纠正这个问题需要两个 UPDATE 语句，如下所示：

```
UPDATE score SET score = 13 WHERE event_id = 5 AND student_id = 8;
UPDATE score SET score = 18 WHERE event_id = 5 AND student_id = 9;
```

你可以把这两行的考试分数修改为正确的值，必须把它们当做一个整体同时修改。第 7 章中的示例增加了几条 SQL 语句，它们设置了自动提交模式、提交、回滚等步骤。在一个使用 DBI 事务处理机制的 Perl 脚本里，考试分数的修改工作将由以下代码来完成：

```
my $orig_re = $dbh->{RaiseError}; # save error-handling attributes
my $orig_pe = $dbh->{PrintError};
my $orig_ac = $dbh->{AutoCommit}; # save auto-commit mode

$dbh->{RaiseError} = 1;           # cause errors to raise exceptions
$dbh->{PrintError} = 0;           # but suppress error messages
$dbh->{AutoCommit} = 0;           # don't commit until we say so

eval
{
  # issue the statements that are part of the transaction
  my $sth = $dbh->prepare (qq{
```

```
                UPDATE score SET score = ?
                WHERE event_id = ? AND student_id = ?
              });
  $sth->execute (13, 5, 8);
  $sth->execute (18, 5, 9);
  $dbh->commit();                 # commit the transaction
};
if ($@)                           # did the transaction fail?
{
  print "A transaction error occurred: $@\n";
  # roll back, but use eval to trap rollback failure
  eval { $dbh->rollback (); }
}

$dbh->{AutoCommit} = $orig_ac;    # restore auto-commit mode
$dbh->{RaiseError} = $orig_re;    # restore error-handling attributes
$dbh->{PrintError} = $orig_pe;
```

eval 语句块负责执行事务，它的终止状态将被保存在变量$@里。如果 UPDATE 语句和 commit()的执行都没有出错，$@将为空；否则，eval 语句块就会终止执行，$@将包含出错信息。如果事务失败，示例代码会在打印出错信息后调用 rollback()方法来撤销事务。注意，为防止因 rollback()调用失败而导致脚本结束，我们把 rollback()调用也放到了它自己的 eval 块里。

在本章里，DBI 脚本所使用的出错处理模式基本上都启用 RaiseError 且禁用 PrintError 属性。也就是说，它们已经有了执行事务所要求的值。因此，上例中用来保存、设置、复原这些属性的代码就显得有点多余。不过，这些“多余的”代码能够确保脚本可以在任何环境中运行，即使你事先不知道出错处理属性的设置情况。

8.3 DBI 脚本实战

在了解了 DBI 程序设计工作中的许多概念之后，我们现在要进入示例数据库去进行一些实战了。第 1 章曾提出了一些任务，我们将通过编写 DBI 脚本来解决一些问题，如下所示。

- 对于考试记分项目，我们希望能检索任意一次给定的考试或测验的分数。
- 对于美国历史研究会这个项目，我们想实现以下几项任务。
  - 按不同格式生成会员名录。我们想为研究会的年会生成一份只包含会员姓名的名单，还想生成一份可供打印的会员名录。
  - 查出哪些会员需要在近期续交会费以保留会员资格，并以电子邮件的形式向他们发出通知。
  - 编辑会员记录。(你总得为那些续交了会费的会员修改资格失效日期吧。)
  - 查找兴趣相同的会员。
  - 把会员名录放到网上去。

在这些任务里，有些可以通过编写在命令行运行的脚本来解决，另外一些则要等到在 8.4 节利用与 Web 服务器配合使用的脚本来解决。到本章末尾，还会有一些目标有待实现，我们将在第 9 章完成那些任务。

8.3.1 生成美国历史研究会会员名录

我们的目标之一是按不同的格式来生成美国历史研究会的名录信息。在各种格式中，最简单的莫

过于一份只包含会员姓名的花名册了，这份花名册将用于生成研究会年会请柬的打印程序。这份花名册可以是普通文本格式。它是生成年会请柬程序的大文档的一部分，只要能把有关信息传递到文档就可以了。

打印版的会员名录就不应该再使用普通文本格式了，它应该有更好的格式。一个合理的选择是 RTF（Rich Text Format），这是微软公司开发的格式，大多数字处理软件都支持它，微软的 Word 自不必说，许多其他软件（如 Open Office）也支持这种格式。不同的字处理器对 RTF 的支持程度会有所不同，但我们将只使用全套 RTF 技术规范的一个基本子集，任何一个支持 RTF 的程序都应该毫无问题地理解它。比如说，在 Mac OS X 系统上，TextEdit 编辑器和 Safari 浏览器都可以正常显示我们将生成的 RTF 输出。

生成年会请柬（普通文本）和 RTF 格式的具体过程本质上是一样的——先用一个查询命令检索数据项，再用一个循环取回各项数据并进行格式设置。既然相似，就没必要把它们写成两个不同的脚本。因此，我决定只编写一个脚本（gen_dir.pl），但这个脚本必须能生成不同格式的输出。这个脚本需要满足以下要求。

(1) 在写出会员项之前，先根据输出格式完成必要的初始化工作：如果将要生成的是普通文本的名录，那就用不着进行特殊的初始化；如果将要生成的是 RTF 版本，就必须编写文档起始控制指令。

(2) 取回各项，根据需要进行格式设置。

(3) 把数据项全部处理完毕之后，进行必要的清理和退出工作：如果是普通文本格式，就用不着进行特殊的处理；如果是 RTF 版，就必须补足一个文档结束控制指令。

因为我们今后还可能需要利用这个脚本来生成其他格式的输出，所以我决定使用一个交换盒来使它可扩展。这个交换盒其实是一个散列，它的每个元素分别对应着一种输出格式。每个元素表明应该调用哪些函数去完成按给定格式生成输出文档的各个步骤，其构成是：一个初始化函数、一个数据写入函数和一个结束清理函数。如下所示：

```
# switchbox containing formatting functions for each output format
my %switchbox =
(
  "text" =>                 # functions for plain text format
  {
    "init"    => undef,     # no initialization needed
    "entry"   => \&text_format_entry,
    "cleanup" => undef      # no cleanup needed
  },
  "rtf" =>                  # functions for RTF format
  {
    "init"    => \&rtf_init,
    "entry"   => \&rtf_format_entry,
    "cleanup" => \&rtf_cleanup
  }
);
```

上面这个分支结构里的每个元素以相应的文档格式名称（text 或 rtf）作为键。我们将要编写的脚本可以让用户在运行该脚本时在命令行上指定自己想要的格式：

```
% ./gen_dir.pl text
% ./gen_dir.pl rtf
```

有了交换盒，就能很容易地增加其他的输出格式。具体步骤如下。

(1) 针对输出生成的不同阶段编写 3 个格式化函数。

(2) 在交换盒里增加一个新元素，定义了格式名称，并指向输出函数。

(3) 当需要以新格式生成输出时，执行 gen_dir.pl 脚本并在命令行上给出新格式的名称。

下面的代码将根据命令行上的第一个参数来选择适当的交换盒元素。如果没有在命令行上给出格式名称或者给出的是无效的格式名称，脚本将生成一条出错信息并把可用的名称列出来。如果你在命令行上给出了一个有效的格式名称，$func_hashref 将指向交换盒里的某个合适的项：

```
my $formats = join (" ", sort (keys (%switchbox)));
# make sure one argument was specified on the command line
@ARGV == 1
  or die "Usage: gen_dir.pl format_type\nAllowable formats: $formats\n";

# determine proper switchbox entry from argument on command line;
# if no entry is found, the format type is invalid
my $func_hashref = $switchbox{$ARGV[0]};

defined ($func_hashref)
  or die "Unknown format: $ARGV[0]\nAllowable formats: $formats\n";
```

散列%switch 中的键就是输出格式的名称，代码将据此选择格式。如果你给出的是一个有效的格式名称，交换盒里的某个项就会得到匹配，它指向输出函数。如果你给出的是一个无效的输出格式名称，交换盒里的项都将得不到匹配。这种安排有两个好处：其一，你不必把格式名称硬编码在格式选择代码里；其二，如果你在今后又往交换盒里添加了新项，这段代码将自动检测出这一情况而不需要修改。

当你在命令行上给出一个有效的格式名称时，上面这段代码将把$func_hashref 的值设置为散列的引用，散列指向生成被选格式的输出的函数。下面这段代码将依次调用初始化函数、取回和打印数据、调用清除函数：

```
# invoke the initialization function if there is one
&{$func_hashref->{init}} if defined ($func_hashref->{init});

# fetch and print entries if there is an entry formatting function
if (defined ($func_hashref->{entry}))
{
  my $sth = $dbh->prepare (qq{
              SELECT * FROM member ORDER BY last_name, first_name
            });
  $sth->execute ();
  while (my $entry_ref = $sth->fetchrow_hashref ("NAME_lc"))
  {
    # pass entry by reference to the formatting function
    &{$func_hashref->{entry}} ($entry_ref);
  }
}

# invoke the cleanup function if there is one
&{$func_hashref->{cleanup}} if defined ($func_hashref->{cleanup});
```

用来取回数据项的循环使用了 fetchrow_hashref()方法。选用这个方法的理由是：如果循环取回的是一个数组，格式函数就必须知道数据列的顺序。可是，利用$sth->{NAME}属性（这个属性给

出的正是数据列返回时的顺序）不是也可以达到目的吗？为什么要多费周折呢？因为通过散列引用，格式函数可以利用`$entry_ref->{col_name}`来指定其想要数据列的值。要是使用 `NAME` 属性的话，就没有这么简单了。`fetchrow_hashref()`方法能用来生成任意格式的输出，因为我们知道我们需要的任何字段都在散列中。

好了，只要再把各输出格式的有关函数（即交换盒数据项中给出的 3 个函数）编写出来，脚本就能交付使用了。

1. 生成普通文本格式的会员名单

文本输出格式用不着初始化和清理工作，只需要有一个输入格式函数（`text_format _entry()`）就够了。这个函数需要一个对会员项的引用，它会把会员姓名打印出来。处理好会员姓名中的后缀部分有点麻烦："Jr."或"Sr."等姓名后缀的前面应该有一个逗号和一个空格，而"II"或"III"之类的姓名后缀的前面却只应该有一个空格。如下所示：

```
Michael Alvis IV
Clarence Elgar, Jr.
Bill Matthews, Sr.
Mark York II
```

字母"I"、"V"、"X"是仅有的表示辈分的罗马数字，这 3 个字母的不同组合可以表示出从第 1 代一直到第 39 代的辈分。如果辈分超出这个范围，就要用到其他数字，但我想这几个字母应该足够了。因此，我们将根据姓名后缀是否与下面这个模板相匹配来决定是否需要加上一个逗号：

```
/^[IVX]+$/
```

在 `text_format_ entry()`函数里，代码将人名中的各个部分按适当顺序放在一起，这部分代码是我们在生成 RTF 格式的会员名录时也会用到的。因此，为避免在 `rtf_format_entry()`函数里重复书写同样的，把这部分代码写成一个辅助函数：

```
sub format_name
{
my $entry_ref = shift;

  my $name = $entry_ref->{first_name} . " " . $entry_ref->{last_name};
  if (defined ($entry_ref->{suffix}))     # there is a name suffix
  {
    # no comma for suffixes of I, II, III, etc.
    $name .= "," unless $entry_ref->{suffix} =~ /^[IVX]+$/;
    $name .= " " . $entry_ref->{suffix}
  }
  return ($name);
}
```

有了这个 `format_name()`函数，用来打印会员姓名的 `text_format_entry()`函数就简单得不能再简单了：

```
sub text_format_entry
{
  printf "%s\n", format_name ($_[0]);
}
```

2. 生成RTF格式的会员名录

与生成年会请柬相比，生成 RTF 格式会员名录的工作要稍微复杂一些。首先，需要为每条记录打

印出更多的信息；其次，为达到预期效果，我们既要给每条记录加上一些 RTF 控制语言，还必须在文档的开头和结尾加上一些控制语言。RTF 文档的最小框架如下所示：

```
{\rtf0
{\fonttbl {\f0 Times;}}
\plain \f0 \fs24
  ...document content goes here...
}
```

文档以“{”开始，以“}”结束。RTF 关键字以“\”开头，文档中的第一个关键字必须是`\rtfn`，其中的 n 是该文档所遵守的 RTF 标准的版本号。根据前面的讨论，我们将使用`\rtf0`来作为文档的第一个关键字。

在文档内，我们还需要给出一个字体表来表明文档内容所使用的字体。字体表由一组列在一对花括号中间、以关键字`\fonttbl` 开头的字体信息构成。在上面给出的最小框架里，字体表把 0 号字体定义为 Times 字体。（这里只需要一种字体。如果你想把文档弄得更花哨一点，可以再增加几种。）

接下来的几条指令设置默认的格式：`\plain` 选择普通格式，`\f0` 选择 0 号字体（我们已经在字体表里把它定义为 Times 字体了），`\fs24` 把字体大小设置为 12 点（`\fs` 后面的数字以半个点为计量单位）。没有必要设置页边距，因为大多数字处理软件都能提供一组合理的默认值。

初始化函数和清理函数负责生成文档框架，如下所示（为了在输出报告里打印出一个反斜线字符“\”，需要在代码里双写为“\\”）：

```
sub rtf_init
{
  print "{\\rtf0\n";
  print "{\\fonttbl {\\f0 Times;}}\n";
  print "\\plain \\f0 \\fs24\n";
}

sub rtf_cleanup
{
  print "}\n";
}
```

输入格式函数负责生成文档内容。为简单起见，我们将把每一项打印在不同的行上，并给每行加上一个标签。如果对应于某个输出行的内容缺失，就不打印这一行。（比如说，如果某位会员没有电子邮件地址，就不需要打印“Email:”行。）因为有些输出行（如“Address:”）由多个数据列（`street`、`city`、`state`、`zip`）里的信息构成，所以这个脚本必须能够应付各种数据值缺失的情况。下面是我们将要使用的输出格式的一个例子：

```
Name: Mike Artel
Address: 4264 Lovering Rd., Miami, FL 12777
Telephone: 075-961-0712
Email: mike_artel@venus.org
Interests: Civil Rights,Education,Revolutionary War
```

这个项的 RTF 表示形式如下所示：

```
\b Name: Mike Artel\b0\par
Address: 4264 Lovering Rd., Miami, FL 12777\par
Telephone: 075-961-0712\par
```

```
Email: mike_artel@venus.org\par
Interests: Civil Rights,Education,Revolutionary War\par
```

要使 Name:行以粗体显示其首尾两端应为\b（开始以粗体显示，它后面要有一个空格）和\b0（结束粗体显示）。会员姓名是用刚才在第 1 小节里的 format_name()函数格式化得到的。我们在各行的末尾加上了一个段落标记（\par），这个标记告诉字处理软件移动到下一行——没有太复杂的东西。这部分代码的主要难点是格式化地址字符串和判断哪些输出行需要打印：

```
sub rtf_format_entry
{
my $entry_ref = shift;

  printf "\\b Name: %s\\b0\\par\n", format_name ($entry_ref);
  my $address = "";
  $address .= $entry_ref->{street}
              if defined ($entry_ref->{street});
  $address .= ", " . $entry_ref->{city}
              if defined ($entry_ref->{city});
  $address .= ", " . $entry_ref->{state}
              if defined ($entry_ref->{state});
  $address .= " " . $entry_ref->{zip}
              if defined ($entry_ref->{zip});
  print "Address: $address\\par\n"
              if $address ne "";
  print "Telephone: $entry_ref->{phone}\\par\n"
              if defined ($entry_ref->{phone});
  print "Email: $entry_ref->{email}\\par\n"
              if defined ($entry_ref->{email});
  print "Interests: $entry_ref->{interests}\\par\n"
              if defined ($entry_ref->{interests});
  print "\\par\n";
}
```

大家不必拘泥于这个特殊的格式化样式。只要修改 rtf_format_entry()函数，就可以改变任何一个字段的打印格式，也就是说，可以随意改变会员名录的打印样式，而这对会员名录的原始格式（即一个字处理文档）来说是相当困难的。

到这里，gen_dir.pl 脚本就编写完成了。现在，只需发出下面的命令，就可以生成普通文本或 RTF 格式的会员名录了：

```
% ./gen_dir.pl text > names.txt
% ./gen_dir.pl rtf > directory.rtf
```

现在，只需再有一个简单的步骤，就能把普通文本的会员名录读出并粘贴到年会程序文档、或者把 RTF 文件读到一个支持 RTF 的程序里。

DBI 使从 MySQL 提取信息的工作变得简单易行，Perl 语言的文本处理能力又简化了这些信息的输出工作。虽然 MySQL 没提供花哨的输出功能，但这并没有多大的关系，因为你现在已经知道怎样把 MySQL 的数据库处理能力与 Perl 等语言出色的文本处理能力集成在一起了。

8.3.2　发出会费催交通知

在使用原始格式（即一个字处理文档）维护美国历史研究会会员名录时，想查出需要通知哪些会

员续交会费的工作将既耗费时间，又容易出错。把这些信息搬到数据库里以后，发出会费催交通知的工作就可以自动化了。可以先查出哪些会员需要续交会费，再通过电子邮件向他们发出会费催交通知。这样，我们就不必通过电话或者邮政信件来联系他们了。

首先查出哪些会员已超过失效时间或应该在指定日期内交纳他们的会费。这个查询将涉及日期计算，但计算工作相当简单。如下所示：

```
SELECT ... FROM member
WHERE expiration < DATE_ADD(CURDATE(), INTERVAL cutoff DAY)
```

其中，cutoff 表示预定的会费交纳宽限天数。这个查询将把那些应该在这个宽限期内交纳会费（或者资格已失效）的会员项检索出来。如果想知道哪些会员已经失去了会员资格，只需把宽限期设置为0即可。

在检索出需要发出会费催交通知的项之后，下一步该怎么办呢？一种方案是从同一个脚本发出电子邮件，但我认为在发出通知之前先看看这份名单比较稳妥。因此，分两步完成这一任务。

(1) 通过 need_renewal.pl 脚本查出哪些会员应该续交会费了。可以核对一下这个脚本生成的名单，或者编辑它，然后再以它为输入进行下一步：发出会费催交通知。

(2) 通过 renewal_notify.pl 脚本发出催交会费的电子邮件。如果某位会员没有电子邮件地址，这个脚本将向你报告，好让你采用其他手段联络这位会员。

第一步，need_renewal.pl 脚本必须查出需要续交会费的会员，这部分代码如下所示：

```
# use default cutoff of 30 days...
my $cutoff = 30;
# ...but reset if a numeric argument is given on the command line
$cutoff = shift (@ARGV) if @ARGV && $ARGV[0] =~ /^\d+$/;

# inform user what cutoff the script is using
warn "Using cutoff of $cutoff days\n";

my $sth = $dbh->prepare (qq{
            SELECT
              member_id, email, last_name, first_name, expiration,
              TO_DAYS(expiration) - TO_DAYS(CURDATE()) AS days
            FROM member
            WHERE expiration < DATE_ADD(CURDATE(), INTERVAL ? DAY)
            ORDER BY expiration, last_name, first_name
          });
$sth->execute ($cutoff);  # pass cutoff as placeholder value

while (my $entry_ref = $sth->fetchrow_hashref ())
{
  # convert undef values to empty strings for printing
  foreach my $key (keys (%{$entry_ref}))
  {
    $entry_ref->{$key} = "" if !defined ($entry_ref->{$key});
  }
  print join ("\t",
              $entry_ref->{member_id},
              $entry_ref->{email},
              $entry_ref->{last_name},
              $entry_ref->{first_name},
```

8

```
                $entry_ref->{expiration},
                $entry_ref->{days} . " days"),
          "\n";
}
```

need_renewal.pl 脚本的输出如下所示（你看到的结果也许与此不同，因为这个结果是以当前日期为依据的，你试用这个脚本时的日期与我写下这段文字时的日期肯定不一样）：

```
89  g.steve@pluto.com       Garner  Steve   2007-08-03  -38 days
18  york_mark@earth.com     York    Mark    2007-08-24  -17 days
82  john_edwards@venus.org  Edwards John    2007-09-12  2 days
```

请注意，有些会员的会费交纳宽限天数是一个负数。这意味着他们已经丧失了会员资格！（当以手工方式维护数据行时，这些会员从你眼皮底下漏了过去。现在，既然已经把这些信息搬到了数据库里，那些“漏网之鱼”就无处藏身了。）

第二步，renewal_notify.pl 脚本以电子邮件的方式发出会费催交通知。为了使 renewal_notify.pl 脚本更容易使用，我们可以让它接受 3 种命令行参数：会员的 ID 编号、电子邮件地址和文件名。数值参数表示会员的 ID 值，包含一个“@”字符的参数表示电子邮件地址。其他任何东西都将解释为文件名，renewal_notify.pl 脚本将从这个文件里读出 ID 编号或电子邮件地址。这样，你既可以通过 ID 编号、也可以通过电子邮件地址来指定会员；而且，你既可以在命令行上直接指定，也可以把它们列在一个文件里。（具体地说，你可以把 need_renewal.pl 脚本的输出保存为一个文件，再把这个文件用作 renewal_notify.pl 脚本的输入。）

对于每一位需要向其发出会费催交通知的会员，renewal_notify.pl 脚本将从他在 member 数据表里的项中提取他的电子邮件地址，然后向这个地址发出一封电子邮件。如果那个项里没有电子邮件地址，renewal_notify.pl 脚本将生成一条警告信息，好让你通过其他手段与这位会员联系。

下面是处理参数的主循环。如果你没有在命令行上给出任何参数，脚本将从标准输入设备上读取输入。如果你在命令行上给出了一些参数，它们将被传递到 interpret_argument()函数并在那儿被归类为 ID 编号、电子邮件地址或者文件名：

```
if (@ARGV == 0)   # no arguments, read STDIN for values
{
  read_file (\*STDIN);
}
else
{
  while (my $arg = shift (@ARGV))
  {
    # interpret argument, with filename recursion
    interpret_argument ($arg, 1);
  }
}
```

read_file()函数负责读入文件的内容（假设该文件应该打开）并查看每一行的第一个字段（如果我们把 need_renewal.pl 脚本的输出馈入 renewal_notify.pl 脚本，每一行将有多个字段，但我们只想查看第一个字段，它应该是一个 ID 编号），如下所示：

```
sub read_file
{
my $fh = shift;   # handle to already-opened file
```

```
my $arg;

  while (defined ($arg = <$fh>))
  {
    # strip off everything past column 1, including newline
    $arg =~ s/\s.*//s;
    # interpret argument, without filename recursion
    interpret_argument ($arg, 0);
  }
}
```

interpret_argument()函数用来对每个参数进行分类以判断它到底是 ID 编号、电子邮件地址、还是文件名。如果是 ID 编号或电子邮件地址，它将查出相应的会员记录并把它传递到 notify_member()函数。要特别注意那些以电子邮件地址指定的会员。有可能出现两位会员使用同一个电子邮件地址的情况（如一对夫妻），我们可不想把会费催交通知发送给一个不相干的人。为避免这种情况，就得根据电子邮件地址去检查与之对应的 ID 编号是否只有一个。如果某个电子邮件地址匹配到一个以上的 ID 编号，因为无法据此断定到底是哪位会员没有交纳会费，所以我们只能在打印出一条警告信息之后忽略它。

如果某个参数既不像数字也不像电子邮件地址，我们将把它视为文件名并继续从中读取输入数据。在这里也必须谨慎从事，为避免陷入无限循环，如果我们已经是在读取某个文件了，就不应该再开始读取另一个文件。

```
sub interpret_argument
{
my ($arg, $recurse) = @_;

  if ($arg =~ /^\d+$/)    # numeric membership ID
  {
    notify_member ($arg);
  }
  elsif ($arg =~ /@/)     # email address
  {
    # get member_id associated with address
    # (there should be exactly one)
    my $stmt = qq{ SELECT member_id FROM member WHERE email = ? };
    my $ary_ref = $dbh->selectcol_arrayref ($stmt, undef, $arg);
    if (scalar (@{$ary_ref}) == 0)
    {
      warn "Email address $arg matches no entry: ignored\n";
    }
    elsif (scalar (@{$ary_ref}) > 1)
    {
      warn "Email address $arg matches multiple entries: ignored\n";
    }
    else
    {
      notify_member ($ary_ref->[0]);
    }
  }
  else                    # filename
  {
```

8

```
        if (!$recurse)
        {
          warn "filename $arg inside file: ignored\n";
        }
        else
        {
          open (IN, $arg) or die "Cannot open $arg: $!\n";
          read_file (\*IN);
          close (IN);
        }
      }
    }
```

notify_member()函数负责完成实际发出会费催交通知的工作。如果某位会员没有电子邮件地址，notify_member()函数就无法发出任何消息，但它会打印出一条警告信息以提醒你需要另想办法与这位会员联系。利用信息中的会员 ID 编号去执行 show_member.pl 脚本，你就能查看到这位会员的完整资料并找出其电话号码和地址。）下面是 notify_member()函数的代码：

```
sub notify_member
{
my $member_id = shift;

  warn "Notifying $member_id...\n";
  my $stmt = qq{ SELECT * FROM member WHERE member_id = ? };
  my $sth = $dbh->prepare ($stmt);
  $sth->execute ($member_id);
  my @col_name = @{$sth->{NAME}};
  my $entry_ref = $sth->fetchrow_hashref ();
  $sth->finish ();
  if (!$entry_ref)                      # no member found!
  {
    warn "NO ENTRY found for member $member_id!\n";
    return;
  }
  if (!defined ($entry_ref->{email}))   # no email address in entry
  {
    warn "Member $member_id has no email address; no message was sent\n";
    return;
  }
  open (OUT, "| $sendmail") or die "Cannot open mailer\n";
  print OUT <<EOF;
To: $entry_ref->{email}
Subject: Your USHL membership is in need of renewal

Greetings.  Your membership in the U.S. Historical League is
due to expire soon.  We hope that you'll take a few minutes to
contact the League office to renew your membership.  The
contents of your member entry are shown below.  Please note
particularly the expiration date.

Thank you.

EOF
```

```
  foreach my $col_name (@col_name)
  {
    printf OUT "$col_name:";
    printf OUT " $entry_ref->{$col_name}"
            if defined ($entry_ref->{$col_name});
    printf OUT "\n";
  }
  close (OUT);
}
```

notify_member()函数发出电子邮件的办法是：打开一个通往 sendmail 程序的管道并把邮件消息的内容推入这个管道。sendmail 程序的路径名在 renewal_notify.pl 脚本的开头被设置为一个参数。sendmail 程序的位置会随系统的不同而不同，所以可能需要对这个路径做相应的修改：

```
# change path to match your system
my $sendmail = "/usr/sbin/sendmail -t -oi";
```

如果你的系统上没有 sendmail 程序，这个脚本就无法正确工作。（比如说，Windows 系统往往不会安装 sendmail 程序。）为应对这种情况，sampdb 发行版本收录了 renewal_notify.pl 脚本的一个替代版本 renewal_notify2.pl，它是用 Mail::sendmail 模块编写出来的，可以在没有安装 sendmail 程序的系统上运行。只要你的系统上安装有 Mail::sendmail 模块，就可以使用那个替代版本。

这个脚本还可以进一步优化。比如说，可以在 member 数据表里增加一个数据列来记录最近一次会费催交通知是在什么时候发出的，然后让 renewal_notify.pl 脚本在它发出会费催交通知的同时更新那个数据列。这有助于使你不会过于频繁地发出会费催交通知。就目前的现状而言，我们假设你运行这个脚本的频率不会超过每月一次。

这两个脚本到现在就全部完成了。你可以按以下步骤来使用它们。

(1) 运行 need_renewal.pl 脚本以生成一份其会员资格已经失效或将在近期失效的会员名单：

```
% ./need_renewal.pl > tmp
```

(2) 查看 tmp，看它有没有不合理的地方。

(3) 如果检查无误，就把它用做 renewal_notify.pl 脚本的输入以发出会费催交通知：

```
% ./renewal_notify.pl tmp
```

如果不想分别通知各个会员，也可以利用 ID 编号或电子邮件地址来通知会员。比如说，下面的命令将通知两位会员：ID 编号为 18 的会员和电子邮件地址是 g.steve@pluto.com 的会员。

```
% ./renewal_notify.pl 18 g.steve@pluto.com
```

8.3.3 会员记录项的编辑修改

在发出会费催交通知之后，那些收到通知的人们中肯定会有续交会费的。如果有人续交了会费，你就得找个办法去修改他的会员资格失效日期。在下一章里，我们将想办法把会员记录的编辑工作放到 Web 上去。但在本节里，我们还是先编写一个命令行脚本（edit_member.pl），使用一个简单的办法（提示你输入会员记录的各个新值）来完成修改任务。它的执行情况如下所示。

- 如果你在调用 edit_member.pl 脚本时没有在命令行上给出任何参数，它将假设你是想创建一个新会员。于是，它将提示你输入这个新会员的初始资料，并创建出一个新项。
- 如果你在调用 edit_member.pl 脚本时在命令行上给出了一个会员 ID 编号，它将假设你是想

修改该会员的个人资料。它将检索记录项的现有内容并提示你修改各数据列的值。如果你在某个数据列里输入了一个新值，它将替换掉该数据列的当前值。如果你直接按下 Enter 键，该数据列里的值将不发生变化。如果你输入了单词“none”，它将清除数据列里的当前值。（如果不知道某位会员的 ID 编号，可以执行 show_member.pl last_name 命令来查出与给定姓氏 last_name 相匹配的记录项里的 ID 编号值。）

如果只是想修改会员的资格失效日期，专门编写一个能对整个项进行修改的脚本就未免有些小题大做。但从另一方面讲，如果想让一位不懂 SQL 的人也能修改记录项里的各个数据列，这样的一个脚本将提供一种简单而通用的解决方案。（作为一个特例，edit_member.pl 脚本不允许修改 member_id 字段。这个字段里的值将在脚本创建新记录时自动生成，一旦生成，就不允许再改变了。）

edit_member.pl 需要了解的第一件事是 member 数据表里的数据列的名字以及它们是否允许被赋值为 NULL。后一个属性是稍后在清除数据列值时需要用到的（如果某个数据列允许使用 NULL 值，我们将把它赋值为 NULL；如果不允许，就把它赋值为一个空字符串）。这些必要的信息可以从 INFORMATION_SCHEMNA 数据库中的 COLUMNS 数据表里查到：

```
my @col_name = ();        # array of column names
my %nullable = ();        # column nullability, keyed on column name
# get member table column names
my $sth = $dbh->prepare (qq{
            SELECT COLUMN_NAME, UPPER(IS_NULLABLE)
            FROM INFORMATION_SCHEMA.COLUMNS
            WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?
          });
$sth->execute ("sampdb", "member");
while (my ($col_name, $is_nullable) = $sth->fetchrow_array ())
{
  push (@col_name, $col_name);
  $nullable{$col_name} = ($is_nullable eq "YES");
}
```

在收集到关于数据列的信息后，这个脚本将按顺序生成一个数组来存放数据列的名字，还将生成一个以数据列的名字为键的散列来标识每个数据列是否允许赋值为 NULL。接下来，edit_member.pl 进入它的主循环：

```
if (@ARGV == 0) # if no arguments were given, create a new entry
{
  # pass reference to array of column names
  new_member (\@col_name);
}
else            # otherwise edit entries using arguments as member IDs
{
  # save @ARGV, and then empty it so that when the script reads from
  # STDIN, it doesn't interpret @ARGV contents as input filenames
  my @id = @ARGV;
  @ARGV = ();
  # for each ID value, look up the entry, and then edit it
  while (my $id = shift (@id))
  {
    $sth = $dbh->prepare (qq{
             SELECT * FROM member WHERE member_id = ?
```

```
            });
        $sth->execute ($id);
        my $entry_ref = $sth->fetchrow_hashref ();
        $sth->finish ();
        if (!$entry_ref)
        {
          warn "No member exists with member ID = $id\n";
          next;
        }
        # pass reference to array of column names and reference to entry
        edit_member (\@col_name, $entry_ref);
      }
    }
```

下面是用来创建一条新会员记录项的代码。它先提示你输入 member 数据表各数据列的值，然后发出一条 INSERT 语句来插入一条新数据行：

```
    sub new_member
    {
    my $col_name_ref = shift; # reference to array of column names
    my $entry_ref = { };    # create new entry as a hash

      return unless prompt ("Create new entry (y/n)? ") =~ /^y/i;
      # prompt for new values; user types in new value, or Enter
      # to leave value unchanged, "NONE" to clear the value, or
      # "EXIT" to exit without creating the record.
      foreach my $col_name (@{$col_name_ref})
      {
        next if $col_name eq "member_id";   # skip key field
        my $col_val = col_prompt ($col_name, undef);
        next if $col_val eq "";             # user pressed Enter
        return if uc ($col_val) eq "EXIT";  # early exit
        if (uc ($col_val) eq "NONE")
        {
          # enter NULL if column is nullable, empty string otherwise
          $col_val = ($nullable{$col_name} ? undef : "");
        }
        $entry_ref->{$col_name} = $col_val;
      }
      # show values, ask for confirmation before inserting
      show_member ($col_name_ref, $entry_ref);
      return unless prompt ("\nInsert this entry (y/n)? ") =~ /^y/i;

      # construct an INSERT query, and then issue it.
      my $stmt = "INSERT INTO member";
      my $delim = " SET "; # put "SET" before first column, "," before others
      foreach my $col_name (@{$col_name_ref})
      {
        # only specify values for columns that were given one
        next if !defined ($entry_ref->{$col_name});
        # quote() quotes undef as the word NULL (without quotes),
        # which is what we want.  Columns that are NOT NULL are
        # assigned their default values.
        $stmt .= sprintf ("%s %s=%s", $delim, $col_name,
```

```
                      $dbh->quote ($entry_ref->{$col_name}));
      $delim = ",";
    }
    $dbh->do ($stmt) or warn "Warning: new entry not created!\n"
  }
```

edit_member.pl 脚本使用了两个例程来提示你输入信息。prompt()函数提出一个问题并返回答案：

```
sub prompt
{
my $str = shift;

  print STDERR $str;
  chomp ($str = <STDIN>);
  return ($str);
}
```

col_prompt()函数接受一个数据列名作为其参数。它先打印出这个数据列名提示你输入新数据列表，再返回你输入的新值：

```
sub col_prompt
{
my ($col_name, $entry_ref) = @_;

  my $prompt = $col_name;
  if (defined ($entry_ref))
  {
    my $cur_val = $entry_ref->{$col_name};
    $cur_val = "NULL" if !defined ($cur_val);
    $prompt .= " [$cur_val]";
  }
  $prompt .= ": ";
  print STDERR $prompt;
  my $str = <STDIN>;
  chomp ($str);
  return ($str);
}
```

col_prompt()函数的第二个参数是一个散列引用，这个散列对应着会员的记录项。如果是创建一个新项，这个值将是 undef；如果是编辑一个现有的项，它将指向该项的当前内容。在后一种情况里，col_prompt()函数将把各数据列的当前值也包含在提示字符串里显示出来，用户将看到它，直接按下 Enter 键即可接受这个值。

用来编辑一个现有会员的代码与用来创建一个新会员的代码很相似。但因为有关的记录项已经存在，所以提示例程还需要把该项的当前值也显示出来；此外，edit_member()函数将发出一条 UPDATE 语句而不是一条 INSERT 语句：

```
sub edit_member
{
# references to an array of column names and to the entry hash
my ($col_name_ref, $entry_ref) = @_;

  # show initial values, ask for okay to go ahead and edit
```

```
    show_member ($col_name_ref, $entry_ref);
    return unless prompt ("\nEdit this entry (y/n)? ") =~ /^y/i;
    # prompt for new values; user types in new value, or Enter
    # to leave value unchanged, "NONE" to clear the value, or
    # "EXIT" to exit without changing the record.
    foreach my $col_name (@{$col_name_ref})
    {
      next if $col_name eq "member_id";   # skip key field
      my $col_val = col_prompt ($col_name, $entry_ref);
      next if $col_val eq "";             # user pressed Enter
      return if uc ($col_val) eq "EXIT";  # early exit
      if (uc ($col_val) eq "NONE")
      {
        # enter NULL if column is nullable, empty string otherwise
        $col_val = ($nullable{$col_name} ? undef : "");
      }

      $entry_ref->{$col_name} = $col_val;
    }
    # show new values, ask for confirmation before updating
    show_member ($col_name_ref, $entry_ref);
    return unless prompt ("\nUpdate this entry (y/n)? ") =~ /^y/i;

    # construct an UPDATE query, and then issue it.
    my $stmt = "UPDATE member";
    my $delim = " SET "; # put "SET" before first column, "," before others
    foreach my $col_name (@{$col_name_ref})
    {
      next if $col_name eq "member_id"; # skip key field
      # quote() quotes undef as the word NULL (without quotes),
      # which is what we want.
      $stmt .= sprintf ("%s %s=%s", $delim, $col_name,
                        $dbh->quote ($entry_ref->{$col_name}));
      $delim = ",";
    }
    $stmt .= " WHERE member_id = " . $dbh->quote ($entry_ref->{member_id});
    $dbh->do ($stmt) or warn "Warning: entry not undated!\n"
  }
```

edit_member.pl 脚本的一个不足之处是它没有检查输入值。member 数据表的大多数字段并不需要检查，它们都是一些字符串字段。但会员资格失效日期字段却需要检查，从而保证输入值是日期。在编写一个通用性数据录入程序时，你应该想提取数据表里的信息，然后以此为依据确定数据列的类型，再根据类型进行检查。输入值的检查是一个很复杂的问题，我不打算在这里做细致深入的讨论。但我决定在 col_prompt()函数增加一些代码以检查 expiration 数据列的输入值格式是否合法。下面是最基本的数据值检查的代码：

```
  sub col_prompt
  {
  my ($col_name, $entry_ref) = @_;

  loop:
    my $prompt = $col_name;
    if (defined ($entry_ref))
```

```
  {
    my $cur_val = $entry_ref->{$col_name};
    $cur_val = "NULL" if !defined ($cur_val);
    $prompt .= " [$cur_val]";
  }
  $prompt .= ": ";
  print STDERR $prompt;
  my $str = <STDIN>;
  chomp ($str);
  # perform rudimentary check on the expiration date
  if ($str && $col_name eq "expiration")  # check expiration date format
  {
    if ($str !~ /^\d+\D\d+\D\d+$/)
    {
      warn "$str is not a legal date, try again\n";
      goto loop;
    }
  }
  return ($str);
}
```

这段代码中的匹配模板测试了 3 种以非数字字符分隔的数字序列。这个检查并不完备，它会把"1999-14-92"之类的值判定为合法值。如果你想让这个脚本更加完善，就需要让它对输入值做更严格的检查，或者再增加一些其他的检查，比如要求人名和姓氏字段不得为空等。

还可以对这个脚本做其他改进。

- 如果用户没有对某条现有记录项进行任何修改，就不发出 UPDATE 语句。这项改进可以这样来实现：先把各数据列的初始值保存起来，然后编写 OPDATE 语句更新被你修改过的数据列，如果你什么都没有改，就不需要发出 UPDATE 语句。
- 如果某条数据行在你正在对之进行修改的期间被别人抢先修改了，脚本将通知你。为此，在 WHERE 子句里为各数据列的原始值分别加上 AND col_name = col_value。这样，一旦有人抢先修改了数据行，你发出的 UPDATE 语句就会失败，表明有两个人同时在试图修改这个项了。
- 启用严格 SQL 模式和其他输入限制，这将使 MySQL 拒绝接受不符合要求的数据值，并在输入数据无法使用时返回一条出错消息，如下所示：

  ```
  $dbh->do ("SET sql_mode = 'TRADITIONAL'");
  ```

edit_member.pl 脚本还有另一个值得改进的地方：它会在进入提示循环之前打开一个数据库连接，并把这个连接保持到那个循环里，完成了对数据行的输出之后才关闭。换句话说，如果用户在输入或修改数据行时花费了很长的时间，或者只是因为临时有事而离开了一会儿，那么在这段时间内，连接将一直处于打开状态。那么，怎样修改 edit_member.pl 脚本才能让它与数据库的连接时间最短呢？

8.3.4　寻找志趣相同的会员

美国历史研究会成员可能希望得到对美国历史事件（如大萧条时期或林肯总统的生平等）有着共同兴趣的其他成员的名单，而为他们提供一份这样的名单正是研究会秘书的职责之一。当会员名录被保存为字处理文档格式时，因为有字处理软件提供的“查找”功能，所以查找这些会员的工作还不算困难。可要想生成一份只包括这些会员在内的名单就不那么容易了，你得使用大量的复制和粘贴操作。

而有了 MySQL 之后，这项工作就变得简单多了，只需执行下面的查询就能达到目的：

```
SELECT * FROM member WHERE interests LIKE '%lincoln%'
ORDER BY last_name, first_name
```

不过，如果在 mysql 客户程序里执行的这个查询，其结果还不怎么好看。下面，用 DBI 脚本 interests.pl 来搜索并生成一份更美观的输出报告。这个脚本先要检查以确认我们在命令行上至少给出了一个参数，如果一个参数都没有，就没什么可查找的了。然后，这个脚本将使用每个参数去搜索 member 数据表的 interests 数据列，如下所示：

```
@ARGV or die "Usage: interests.pl keyword\n";
search_members (shift (@ARGV)) while @ARGV;
```

在搜索关键字字符串时，先在它的两端分别加上一个“%”通配符以构造出一个匹配模板，然后进行模式匹配。这样，不管这个字符串出现在 interests 数据列里的什么位置，都能把它找出来。然后，打印匹配到的记录项：

```
sub search_members
{
my $interest = shift;

  print "Search results for keyword: $interest\n\n";
  my $sth = $dbh->prepare (qq{
              SELECT * FROM member WHERE interests LIKE ?
              ORDER BY last_name, first_name
            });
  # look for string anywhere in interest field
  $sth->execute ("%" . $interest . "%");
  my $count = 0;
  while (my $hash_ref = $sth->fetchrow_hashref ())
  {
    format_entry ($hash_ref);
    ++$count;
  }
  print "Number of matching entries: $count\n\n";
}
```

format_entry()函数负责把一个记录项转换为它的可打印形式。这个函数与 gen_dir.pl 脚本里的 rtf_format_entry()函数基本一致，只是去掉了后者中的的 RTF 控制字而已，所以它的代码就不在这里重复给出了。如果你想了解它的具体实现，请查阅 sampdb 发行版本里的 interests.pl 脚本。

8.3.5 把会员名录放到网上

在 8.4 节，我们将开始编写脚本通过显示在客户端 Web 浏览器里的 Web 页面去连接 MySQL 服务器以提取和显示信息。那些脚本将根据客户请求去动态地生成 HTML 文档。在此之前，先编写一个能够生成一份静态 HTML 文档的 DBI 脚本来，生成的 HTML 文档可以加载到 Web 服务器的文档树里来查看其内容了。我们现在的任务是生成一份 HTML 格式的美国历史研究会会员名录（别忘了，把会员名录放到网上是目标之一）。

下面是 HTML 文档的一个基本框架：

```
<html>                              文档的开始标记
<head>                              文档头的开始标记
<title>My Page Title</title>        文档的标题
</head>                             文档头的结束标记
<body bgcolor="white">              文档体的开始标记（白色背景）
<h1>My Level 1 Heading</h1>         一个一级小标题

文档体的内容

</body>                             文档体的结束标记
</html>                             文档的结束标记
```

按 HTML 格式生成这个目录并不需要编写一个全新的脚本。别忘了，我们此前编写 gen_dir.pl 脚本时使用了一个可扩展的脚本框架，可以把按其他格式生成目录的代码填充到那个框架里。既然如此，我们现在就来享受一下可扩展性的好处，把用来生成 HTML 格式的目录的代码添加进去吧。为了完成这项工作，需要对 gen_dir.pl 脚本做以下修改。

- 编写文档的初始化函数和清理函数。
- 编写一个用来对各成员数据行格式化的函数。
- 增加一个用来标识格式名称的交换盒元素，并把这个元素与用来生成那种格式的输出的函数关联起来。

上面给出的 HTML 文档基本框架可以被相当容易地划分为开头、中间、结尾 3 个部分。开头和结尾部分对应着我们将要编写的初始化函数和清理函数，而中间部分则由输入格式函数来生成。HTML 初始化函数将负责生成从开始标记开始的所有内容，而清理函数将负责生成</body>和</html>标记。如下所示：

```
sub html_init
{
  print "<html>\n";
  print "<head>\n";
  print "<title>U.S. Historical League Member Directory</title>\n";
  print "</head>\n";
  print "<body bgcolor=\"white\">\n";
  print "<h1>U.S. Historical League Member Directory</h1>\n";
}

sub html_cleanup
{
  print "</body>\n";
  print "</html>\n";
}
```

和往常一样，主要工作都将集中于对输入进行格式化，但这部分代码并不难写：把 rtf_format_entry()函数复制过来并命名为 html_format_entry()，修改它，使得记录项里的每一个特殊字符都被编码，最后再把 RTF 控制字替换为 HTML 标记就行了。如下所示：

```
sub html_format_entry
{
my $entry_ref = shift;

  # Convert <, >, ", and & to the corresponding HTML entities
```

```
  # (&lt;, &gt;, &quot, &)
  foreach my $key (keys (%{$entry_ref}))
  {
    next unless defined ($entry_ref->{$key});
    $entry_ref->{$key} =~ s/&/&/g;
    $entry_ref->{$key} =~ s/\"/&quot;/g;
    $entry_ref->{$key} =~ s/>/&gt;/g;
    $entry_ref->{$key} =~ s/</&lt;/g;
  }
  printf "<strong>Name: %s</strong><br />\n", format_name ($entry_ref);
  my $address = "";
  $address .= $entry_ref->{street}
              if defined ($entry_ref->{street});
  $address .= ", " . $entry_ref->{city}
              if defined ($entry_ref->{city});
  $address .= ", " . $entry_ref->{state}
              if defined ($entry_ref->{state});
  $address .= " " . $entry_ref->{zip}
              if defined ($entry_ref->{zip});
  print "Address: $address<br />\n"
              if $address ne "";
  print "Telephone: $entry_ref->{phone}<br />\n"
              if defined ($entry_ref->{phone});
  print "Email: $entry_ref->{email}<br />\n"
              if defined ($entry_ref->{email});
  print "Interests: $entry_ref->{interests}<br />\n"
              if defined ($entry_ref->{interests});
  print "<br />\n";
}
```


下面是这个函数生成的输出：

```
<strong>Name: Mike Artel</strong><br />
Address: 4264 Lovering Rd., Miami, FL 12777<br />
Telephone: 075-961-0712<br />
Email: mike_artel@venus.org<br />
Interests: Civil Rights,Education,Revolutionary War<br />
<br />
```

遵照 XHTML 标准，我们使用了
标记而非
标记。XHTML 的语法比 HTML 更严格，它们之间的主要区别将在 8.4.2 节中的第 2 小节里介绍。

还需要对 gen_dir.pl 脚本做最后一项修改：在交换盒里增加一个与 HTML 格式函数相对应的新元素。下面是完成修改后的交换盒代码，它的最后一个元素定义了一个名为 html 的格式，它指向那几个用来生成 HTML 格式文档各部分的函数：

```
# switchbox containing formatting functions for each output format
my %switchbox =
(
  "text" =>                 # functions for plain text format
  {
    "init"    => undef,     # no initialization needed
    "entry"   => \&text_format_entry,
    "cleanup" => undef      # no cleanup needed
```

```
    },
    "rtf" =>                   # functions for RTF format
    {
      "init"    => \&rtf_init,
      "entry"   => \&rtf_format_entry,
      "cleanup" => \&rtf_cleanup
    },
    "html" =>                  # functions for HTML format
    {
      "init"    => \&html_init,
      "entry"   => \&html_format_entry,
      "cleanup" => \&html_cleanup
    }
  );
```

现在，只要执行下面这条命令并把结果输出文件 directory.html 添加到 Web 服务器的文档树里就可以得到一份 HTML 格式的会员名录了：

```
% ./gen_dir.pl html > directory.html
```

无论何时修改数据库里的 member 数据表，都需要再次运行这条命令以更新在线版会员名录。如果不想以手动执行这条命令，可以考虑把它设置成能定期自动更新在线版会员名录。在 Unix 上，可以使用 cron 来完成。假设 gen_dir.pl 脚本被安装在/usr/local/bin 子目录里，美国历史研究会在 Web 服务器文档树里的子目录是/usr/local/apache/htdocs/ushl，那么下面这条 crontab 项将在每天凌晨 4 点更新会员名录（在一行上输入整条命令）：

```
0 4 * * * /usr/local/bin/gen_dir.pl
  > /usr/local/apache/htdocs/ushl/directory.html
```

如果想使用这个 cron 项，就必须拥有将文件写入文档树的权限。

8.4　用 DBI 开发 Web 应用

我们此前开发的 DBI 脚本都是为了在命令行环境使用，但 DBI 同样能用在其他上下文里，比如用在基于 Web 的应用程序开发中。如果你编写的 DBI 脚本能够让 Web 服务器根据 Web 浏览器发来的请求来执行它们，就等于是在用户和你的数据库之间开辟了一个新的互动舞台。比如说，如果你编写了一个以表格形式显示数据的脚本，把每个数据列的标题转换为一个链接并不难。而当用户点击这些链接时，数据列里的数据将重新排序。这样一来，用户只要轻点鼠标就可以换个方式查看数据而无需输入任何查询。你还可以提供一个表单，用户在这个表单里输入各种数据库检索条件，再把检索结果放到一个网页里返回给用户。这些做法虽然简单，却能大幅改善数据库访问操作的交互性。此外，因为 Web 浏览器的显示能力通常要优于终端窗口，所以你还能创造出更美观的输出效果来。

在本节里，我们将编写以下几个基于 Web 的脚本。

- 为 sampdb 数据库编写一个通用性的数据表浏览器。这个脚本与我们计划中的数据库应用任务没有任何关系。但它将演示 Web 程序设计工作中的几个概念，并提供一个方便的数据表内容查看工具。
- 一个成绩浏览器，我们可以用它任意查看某次考试或测验的成绩。这不仅为成绩记录项目的考试分数核对工作提供了方便，还可以用来为每次考试制作考试分数分布曲线，从而更准确地在考卷上用字母标出分数等级。

- ❑ 一个用来寻找有共同爱好的历史学会成员的脚本。在使用时，先让用户输入一个搜索范围，然后在该范围内搜索 `member` 数据表的 `interests` 数据列。其实我们在前面的 8.3.4 节里已经编写了一个名为 interests.pl 的命令行脚本来做这件事，但这个命令行版本只有那些在安装着该脚本的机器上有登录账户的人才能执行。增加一个基于 Web 的版本将使这个目录对每一个有 Web 浏览器的人开放。新增版本的另一个好处是建立起了一种参照体系，让我们可以对完成同一任务的不同方法进行多方面的对照评比。（事实上，我们开发了两个基于 Web 的实现。一个基于模式匹配功能，就像 interests.pl 脚本那样。另一种使用的是 `FULLTEXT` 搜索功能。）

编写这些脚本要用到 Perl 的 CGI.pm 模块，它提供了一套能够把 DBI 连接到 Web 的简单机制。（CGI.pm 模块的获得和安装办法见附录 A。）利用 CGI.pm 模块编写出来的 DBI 脚本能够使用 CGI（Common Gateway Interface protocol，通用网关协议，它对 Web 服务器与其他程序的通信作出了定义）——这个模块的名字也正是由此而来。CGI.pm 模块将负责完成许多普通的内务工作，比如收集 Web 服务器传递给你脚本的参数值（作为输入）。CFI.pm 还为生成 HTML 格式的输出准备了几个方便的方法，它们有助于降低 HTML 文档出错的概率，因为通过它们写出来的 HTML 标记肯定不会像自己写的那么容易出现笔误。

虽说本章对 CGI.pm 模块的介绍已经足以让你编写出自己的 Web 应用程序，但我们不可能在这里把 CGI.pm 模块的每个功能都介绍到。如果你想进一步了解它，可以去阅读由 Lincoln Stein 撰写的 *Official Guide to Programming with CGI.pm*（John Weiley，1998）或者查阅下面这个网址上的在线文档：

http：//stein.cshl.org/ WWW/ software/ CGI/

我的另外一本书 *MySQL and DBI for Web*（New Rider，2000）也介绍了 CGI.pm，它是专门讨论 MySQL 与 DBI 的。

在本章后续内容里介绍的基于 Web 的脚本都可以在 `sampdb` 发行版本中的/perlapi/web 目录里找到。

8.4.1 配置Apache服务器使用CGI脚本

要想开发基于 Web 的脚本，光有 DBI 和 CGI.pm 模块是不够的，你还必须安装 Web 服务器。这里的讨论重点将集中在 Apache 服务器与 DBI 脚本的配合使用方面，但只要你懂得变通并能灵活运用有关规则，也完全可以使用另外一种服务器。

在下面的讨论里，假设你已经把 Apache 服务器软件安装在子目录/usr/local/apache（如果使用的是 Unix 系统）或者 C:\ Apache（如果使用的是 Windows 系统）里了。在 Apache 软件的顶级目录之下，最重要的子目录是 htdocs（HTML 文档树）、cgi-bin（用来存放将由 Apache 服务器调用的可执行的脚本和程序）和 conf（用来存放配置文件）。你的系统可能会把这些子目录安排在其他位置，如果真是这样，请对以下内容做相应的调整。

应该确保 cgi-bin 子目录没有出现在 Apache 的文档树中。这项预防性安全措施可以防止客户有意或无意地请求查看脚本的源代码文本。你肯定不想让恶意客户有机会弄到脚本的源代码并对它们进行分析，然后利用在其中发现的安全漏洞去攻击你的系统。

要安装供 Apache 服务器调用的 CGI 脚本，就需要把它复制到 cgi-bin 子目录。在 Unix 系统上，脚本的第一行必须以“#!”开头，脚本本身也必须被设置为可执行模式，就像命令行脚本一样。此外，最好把这个脚本的属主设置为将用来运行 Apache 服务器的用户并只允许这位用户访问。比如说，如果 Apache 服务器将由一个名为“www”的用户运行，下面两条命令将把脚本 `myscript.pl` 的属主设

置为“www”并只允许这位用户执行和读取这个脚本：

```
# chown www myscript.pl
# chmod u=rx,go-rwx myscript.pl
```

你可能需要以 root 身份来执行这两条命令。如果你没有把脚本安装到 cgi-bin 子目录里去的权限，请找系统管理员来帮忙完成。

如果你使用的是 Windows 系统，chown 和 chmod 命令就不需要了，但脚本的第一行仍应以“#!”开头。这一行将列出 Perl 程序的完整路径名。比如说，如果把 Perl 安装为 C:\Perl\bin\perl.exe，就应该把“#!”行写成下面这个样子：

```
#!C:/Perl/bin/perl
```

在 Windows 系统上，还有一个更简单的办法：把 Perl 的路径名添加到环境变量 PATH 里。如果你已经这样做了，就可以把脚本的第一行写成下面这个样子：

```
#!perl
```

sampdb 发行版本里的 Perl 语言脚本都在“#!”行里把 Perl 的路径名写成/usr/bin/perl。如果你把 Perl 安装在其他地方，就需要对每个脚本做相应的修改。

把脚本安装到 cgi-bin 目录里之后，你就可以通过从 Web 浏览器向 Web 服务器发出相应的 URL 来请求它了。比如说，如果 Web 服务器运行在本地主机上，你就可以使用下面这个 URL 来请求 myscript.pl 脚本：

```
http://localhost/cgi-bin/myscript.pl
```

在你自己的脚本里，千万不要忘记把本章各示例程序中的 URL 地址改成指向你自己的 Web 服务器主机，而不是指向 localhost。

用 Web 浏览器来请求脚本将导致它被 Web 服务器调用执行，该脚本的执行结果将被 Web 服务器送回到 Web 浏览器并被显示为一个页面。

当你从命令行执行 DBI 脚本时，警告和出错信息都将被送往终端。但因为 Web 环境根本没有终端，所以这些信息都将被送往 Apache 的出错日志。这些信息对脚本的调试工作有着重要的帮助意义，所以一定要把这个日志的存放地点找出来。在我的系统上，它是 Apache 根目录/usr/local/apache 下的 logs 子目录里的 error_log 文件。你系统上的情况可能与我的不同。这个日志的存放地点是由配置文件 httpd.conf（位于 Apache 的 conf 子目录）里的 Errorlog 选项设定的。

8.4.2　CGI.pm模块简介

如果你在编写 Perl 脚本时会用到 CGI.pm 模块，就必须在脚本的开头放上一条 use CGI 语句以导入模块里的函数名。下面这条语句将导入最常用的那些函数构成的标准集：

```
use CGI qw(:standard);
```

有了这条语句，你就可以调用 CGI.pm 模块中的函数来生成各种 HTML 结构了。一般说来，那些函数的名字与相应的 HTML 元素是一致的。比如说，如果想生成一个一级标题和一个段落，就需要调用 h1()和 p()函数，如下所示：

```
print h1 ("This is a header");
print p ("This is a paragraph");
```

CGI.pm 还支持面向对象的编程风格，你可以在事先没有导入函数名的情况下调用这个模块里的函数。具体做法是：增加一条 use 语句并创建一个 CGI 对象：

```
use CGI;
my $cgi = new CGI;
```

然后就可以通过这个 CGI 对象来调用 CGI.pm 模块里的函数了。此时，那些函数将被视为这个 CGI 对象的方法：

```
print $cgi->h1 ("This is a header");
print $cgi->p ("This is a paragraph");
```

如果使用的是面向对象的接口，就必须写出$cgi->这个前缀。为简洁起见，本书里将使用较简单的函数调用接口。但使用函数调用接口有这样一个弊端：一旦 CGI.pm 模块里的函数与 Perl 语言中的内建函数发生同名现象，你就不得不另想一种不会产生冲突的办法来调用。比如说，CGI.pm 模块里有一个名为 tr()的函数，它将生成一对放在 HTML 表格里数据行各单元两端的<tr>和</tr>标记。这个函数与 Perl 语言中用于翻译的内建函数 tr 出现了名字冲突。为了解决这一矛盾，必须在使用 CGI.pm 模块的函数调用接口时把 tr()写成 Tr()或 TR()。如果使用的是面向对象的接口，就不会出现这种问题。因为 tr()是作为$cgi 对象的一个方法调用的，所以你在代码里得用$cgi->tr()形式来调用它，这就能与 Perl 语言内建的函数名区分开了。

1. 读取来自Web的输入参数

CGI.pm 模块能替你收集 Web 服务器传递给脚本的输入信息，你只要调用 param()函数就可以获得信息。如果想知道所有可用参数的名字，可以使用下面这条语句：

```
my @param = param ();
```

如果想检索某个参数的值，把它的名字传递给 param()函数就行了。如果该参数有值，param()函数将返回它的值；否则，返回 undef。如下所示：

```
my $my_param = param ("my_param");
print "my_param value: ", (defined ($my_param) ? $my_param : "not set"), "\n";
```

2. 生成Web输出

CGI.pm 模块里的很多函数都是为了生成将送往客户浏览器的输出。请看下面这段 HTML 文档：

```
<html>
<head>
<title>My Simple Page</title>
</head>
<body bgcolor="white">
<h1>Page Heading</h1>
<p>Paragraph 1.</p>
<p>Paragraph 2.</p>
</body>
</html>
```

下面这个脚本将使用 CGI.pm 输出函数生成一份等效的文档：

```
#!/usr/bin/perl
# simple_doc.pl - produce simple HTML page

use strict;
```

```
use warnings;
use CGI qw(:standard);

print header ();
print start_html (-title => "My Simple Page", -bgcolor => "white");
print h1 ("Page Heading");
print p ("Paragraph 1.");
print p ("Paragraph 2.");
print end_html ();
```

header()函数负责生成一个 Content-Type:标头，它必须出现在网页的最开头。如果网页是通过脚本生成的，这个标头就必不可少，它的作用是让浏览器知道紧随其后的文档属于哪一种类型。（这与我们编写静态 HTML 网页时的做法稍有不同。静态 HTML 网页不需要有这个标头，因为 Web 服务器会自动发送一个给浏览器。）在默认的情况下，header()函数将写出一个如下所示的标头来：

```
Content-Type: text/html
```

header()函数的后面就是那些用来生成网页内容的函数调用了。start_html()函数负责生成从<html>开始标记直到<body>开始标记的各种标记，h1()和 p()函数负责写出标题和段落内容，end_html()函数负责写出文档的结束标记。

很多 CGI.pm 函数，如刚才的 start_html()函数，都允许使用一些给定参数，它们的格式都是-name => value。这种安排对带很多可选参数的函数特别有利，因为你不仅可以只列出你需要用到的参数，还可以按任意顺序来列出它们。

虽然 CGI.pm 模块为我们提供了输出生成函数，但你仍可根据具体情况编写一些原始的 HTML。你可以把这两种做法结合起来使用，让脚本里既有对 CGI.pm 函数的调用语句，又有生成文本标记的打印语句。不过，与亲自编写 HTML 相比，使用 CGI.pm 函数来生成输出的一个优势是，有助于你把注意力集中到逻辑单元而非各个标记上，同时， HTML 也不太容易出现错误。（注意，我说的是“不太容易出现错误”，因为 CGI.pm 模块本身并不能阻止你做出一些奇怪的事情，比如在标题里放上一个列表。）

CGI.pm 模块还使脚本具有更好可移植性，这是你无法通过编写 HTML 做到的。比如说，从 2.69 版开始，CGI.pm 将自动生成 XHTML 格式的输出。如果使用老版本的 CGI.pm 模块来编写普通的 HTML，那么，只需升级 CGI.pm 模块的版本，你那些脚本也将开始生成 XMTML 格式的文档。

XHTML 与 HTML 很相似，但有一个更加完善的格式定义。HTML 很容易学习和使用，但有一个问题是，不同的浏览器对 HTML 往往有着不同的解释，例如，对不规范的 HTML 文档有着不同的容忍性。所以，在这个浏览器里能正确显示的网页到了那个浏览器里往往就不能正确显示了。XHTML 的要求就严格得多，这就使文档更加规范。下面是 HTML 和 XHTML 的几个主要区别。

- 与 HTML 不同，XHTML 要求文档中的每一个开始标记必须有一个配对的结束标记。比如说，段落本应该被放在<p>和</p>标记之间，但 HTML 文档却往往允许省略</p>标记。对于没有任何主体部分 HTML 标记，例如
和<hr>等，XHTML 要求它们必须自我封闭，也就是说，应该把它们写成
</br>和<tr></tr>的样子。为解决这一问题，XHTML 也允许使用单个的
和<hr/>标记来同时充当开始标记和结束标记。不过，因为某些早期的浏览器不能正确地解释这类标记，以为是 br/和 hr/，所以你最好把它们写成
和<hr />——在斜线前面加一个空格——以减少这种错误的发生。
- HTML 不区分标记和属性名称中的字母大小写情况。比如说，HTML 会把<BODY>BGCOLOR=

'white'和<body bgcolor="white">看做是同样的标记。XHTML 要求标记和属性名称必须使用小写字母，所以你只能把这个标记写成<body bgcolor="white">。

- 在 HTML 中，属性值可以不在引号内，甚至可以没有。比如说，下面这样的数据单元构造在 HTML 里就是合法的：

```
<td width=40 nowrap>Some text</td>
```

在 XHTML 中，属性必须有值，而且必须放在引号中。对于那些使用时不带值的 HTML 属性，按照惯例，XHTML 把它们的名字用作它们的值。在 XHTML 文档里，上面那个<tr>等效于下面的样子：

```
<td width="40" nowrap="nowrap">Some text</td>
```

本书里所有 Web 脚本生成的输出都遵守 XHTML 规范。在本章里，我们将依靠 CGI.pm 模块去生成格式正确的 XHTML 标记。第 9 章也生成 XHTML，但要靠它们自己去生成各种标记，因为 PHP 不像 CGI.pm 模块那样拥有生成标记的函数。

3. 对HTML和URL文本进行转义

如果你准备写到 Web 页面里去的文本可能包含特殊字符，就应该使用 escapeHTML()函数对这段文字进行处理，以确保那些特殊字符都能得到正确的转义。如果你构造出来的 URL 可能包含特殊字符，也得对它进行转义处理，但此时应该使用 escape()函数。这两个函数眼里的特殊字符是不同的，编码方式也不同，所以你必须选用正确的编码函数。escapeHTML()函数的用途是对 HTML 特殊字符进行必要的转义。比如说，把<转义成<，escape()函数将把一个特殊字符转义为一个以“%”符号引导的两位十六进制数字，该数字代表着数值字符编码；例如<将被转义为%3C。下面这个简短的 Perl 脚本 escape_demo.pl 演示了上述两种转义操作：

```
#!/usr/bin/perl
# escape_demo.pl - demonstrate CGI.pm output-encoding functions

use strict;
use warnings;
use CGI qw(escapeHTML escape);  # import escapeHTML() and escape()

# Assign default string value, but use command-line argument if present
my $s = "1<=2, right?";
$s = shift (@ARGV) if @ARGV;
print "Unencoded string:             ", $s, "\n";
print "Encoded for use as HTML text: ", escapeHTML ($s), "\n";
print "Encoded for use in a URL:     ", escape ($s), "\n";
```

这个脚本分别使用刚才介绍的那两个函数对字符串$s 进行了编码并把结果打印了出来。当你运行这个脚本时，它将产生如下所示的输出。可以清楚地看到，字符串$s 作为 HTML 文本时的编码与它作为 URL 时的编码是不同的：

```
unencoded string:             1<=2, right?
encoded for use as HTML text: 1&lt;=2, right?
encoded for use in a URL:     1%3C%3D2%2C%20right%3F
```

如果你向 escape_demo.pl 脚本提供了一个命令行参数，这个脚本将对那个参数而不是默认字符串进行编码。这使你可以看到自己选择的字符串的转义编码情况。

escape_demo.pl 脚本通过一条 use CGI 语句导入了编码函数的名称。注意，根据 CGI.pm 模块

当前版本的情况，其标准函数集可能并不包括这两个编码函数，所以即使你已经导入了标准函数集，也必须另外导入这两个函数。如下所示：

```
use CGI qw (:standard escapeHTML escape);
```

4. 生成多用途网页

为什么要用基于 Web 的脚本来生成 HTML 页面而不去编写静态的 HTML 文档呢？一个重要的原因是脚本能够根据不同的调用方式生成不同的页面。后面将要编写的 CGI 脚本都具有这一特性，它们都将按以下策略执行。

- 当你从你的浏览器第一次请求脚本时，它会生成一个初始页面供你选择你想要的信息。
- 当你作出选择后，你的浏览器将向 Web 服务器发出一个请求，导致刚才的脚本被再次调用。这一次，脚本将把你请求的详细信息从数据库里检索出来并显示在第二个页面里。

要想实现这一构想，就必须解决这样一个问题：你想根据第一个页面里作出的选择来确定第二个页面里的内容，但 Web 页面通常是彼此无关的，除非作出特殊的安排。一种解决方案是让脚本在生成页面时把某个参数设定为一个特定的值，告诉脚本你想让它在下一次被调用的时候干些什么。第一次调用这个脚本时，那个参数还没有值，于是脚本将生成一个初始页面。当你指定要看的信息之后，这个脚本将再次被调用，不过这次参数被设定为一个值告知脚本应完成什么任务。

让 Web 页面向脚本传递参数值的办法有好几种。一种办法是在页面里提供一个表单让用户填写。用户提交这个表单时，它的内容就会被传递到 Web 服务器。Web 服务器再把信息传递给脚本，脚本通过调用 `param()` 函数就能知道提交的内容。我们为美国历史研究会进行的关键字搜索就将用这个办法来实现：搜索页面里有一个表单，供会员在其中填写他们想搜索的关键字。

向脚本传递参数值的另一种办法是把参数值追加在 URL 尾部，当你向 Web 服务器请求脚本时，那些参数值也将被传递给 Web 服务器。我们用来实现 `sampdb` 数据表浏览器和考试分数浏览器的脚本将以这个办法来实现。向脚本发出指令的另一个办法是在请求该脚本的时候在发往 Web 服务器的 URL 地址的末尾添加参数值。我们将使用这个办法来实现我们的 `sampdb` 数据表浏览器和考试成绩浏览器脚本。这一思路的关键在于用脚本生成一个包含着超链接的页面。当用户点击某个链接时，脚本将会再次执行，而隐藏在那个链接里的一个参数将告诉脚本应该做些什么。实际上，这等于是让脚本换个方式去调用它自己，并根据用户所点击的链接而相应地提供另一种结果。

为让脚本调用它自己，需要在该脚本发送给浏览器的页面里加上一个自引用超链接，即一个指向它自己的 URL 地址的链接。比如说，如果脚本 `myscript.pl` 已经被安装到了 Web 服务器的 cgi-bin 子目录，它所生成的页面里会有一个如下所示的链接：

```
<a href="/cgi-bin/myscript.pl">Click Me!</a>
```

那么，当你在浏览器里点击该页面中的文本“Click Me !”时，浏览器会向 Web 服务器发回一条执行 `myscript.pl` 脚本的请求。当然了，因为上例中的 URL 地址没有任何其他信息，所以它只能做到让脚本再次送回同样的页面。可是，如果你给它加上了一个参数，那个参数也将在你点击这个链接时被送回给 Web 服务器。接下来，当 Web 服务器调用这个脚本时，这个脚本就可以通过 `param()` 函数检测到该参数的取值情况并据此采用相应的行动。

把一个参数值追加到 URL 地址字符串末尾的办法是：先写出一个问号字符，再以 `name = value` 的形式给出这个参数的名字和值。比如说，如果你想添加一个值为 `large` 的 `size` 参数，就得把 URL 地址写成下面的样子：

```
/cgi-bin/myscript.pl?size=large
```

如果你需要传递多个参数，就需要用;或&字符将它们分开：

```
/cgi-bin/myscript.pl?size=large;color=blue
```

CGI.pm 模块还能把;或&字符识别为参数分隔符。不同程序设计语言的 Web 编程 API 有着不同的惯例，所以在构造 URL 地址之前一定要搞清楚它们能够识别的是;还是&。这里使用的是：

如果你想在脚本里构造出一个带有参数且指向该脚本自身的 URL 地址，就需要在脚本里先使用 CGI.pm url()函数得到它自身的 URL 地址，再把参数追加到末尾，如下所示：

```
$url = url ();          # get URL for script
$url .= "?size=large";  # add first parameter
$url .= ";color=blue";  # add second parameter
```

之所以要调用 url()函数来获得脚本的路径，是为了避免把路径硬编码在代码里。

接着，把这个 URL 地址送入 CGI.pm 模块中的 a()函数以生成一个超链接：

```
print a ({-href => $url}, "Click Me!");
```

这条 print 语句将生成一个如下所示的超链接：

```
<a href="/cgi-bin/url.pl?size=large;color=blue">Click Me!</a>
```

上面这个例子在构造$url 值时并没有考虑到参数值或链接标签里可能会有特殊字符的情况。为保险起见，还是用 CGI.pm 模块的编码函数对那些参数值和链接标签进行一下处理为好，除非你能百分之百地肯定不需要对它们进行任何编码。将出现在 URL 地址末尾的值应该用 escape()函数来编码，将出现在普通 HTML 文本里的值应该用 escapeHTML()函数来编码。就拿上面那个例子来说吧，如果你把超链接标签的值保存在$label 变量里，把参数 size 和 color 的值保存在$size 和$color 变量里，就应该像下面这样对它们编码：

```
$url = sprintf ("%s?size=%s;color=%s",
                url (), escape ($size), escape ($color));
print a ({-href => $url}, escapeHTML ($label));
```

下面这个简短的 CGI 脚本 flip_flop.pl，演示了自引用 URL 地址的使用方法。在这个脚本第一次被调用时生成的页面（我们不妨称之为“页面 A”）里有一个超链接，单击超链接将导致 Web 服务器再次调用 flip_flop.pl 脚本，这个链接也包含一个 paged 参数，让脚本生成页面 B。页面 B 里也有一个指向该脚本自身的超链接，但这个超链接不带 pageb 参数。这就意味着如果你点击了页面 B 里的超链接，就会重新看到页面 A。换句话说，连续调用 flip_flop.pl 脚本将使它交替生成页面 A 和页面 B：

```
#!/usr/bin/perl
# flip_flop.pl - simple multiple-output-page CGI.pm script

use strict;
use warnings;
use CGI qw(:standard);

my $url;
my $this_page;
my $next_page;
```

8

```
# determine which page to display based on absence or presence
# of the pageb parameter

if (!defined (param ("pageb"))) # display page A w/link to page B
{
  $this_page = "A";
  $next_page = "B";
  $url = url () . "?pageb=1";
}
else                            # display page B w/link to page A
{
  $this_page = "B";
  $next_page = "A";
  $url = url ();
}

print header ();
print start_html (-title => "Flip-Flop: Page $this_page",
                  -bgcolor => "white");
print p ("This is Page $this_page. To select Page $next_page, "
         . a ({-href => $url}, "click here"));
print end_html ();
```

把这个脚本安装到你的 cgi-bin 子目录，再从你的浏览器里用下面这个 URL 地址请求它，用你自己的 Web 服务器的名字替代 localhost：

```
http://localhost/cgi-bin/flip_flop.pl
```

你不妨在交替出现的页面 A 和页面 B 里的超链接上多单击几次，看 flip_flop.pl 脚本到底是如何交替生成这两个页面的。

现在，如果有另外一个客户也开始请求 flip_flop.pl 脚本，会发生什么样的事情呢？你们会彼此干扰吗？答案是不会。因为你们在首次请求这个脚本时都没有给出 pageb 参数，所以你们首次看到的页面将都是这个脚本所生成的页面 A。而你们此后发出的请求是否带有 pageb 参数则取决于你们看到的当前页面，所以 flip_flop.pl 脚本也将为你们分别生成相应的页面而根本不会干扰到另外一个客户。

8.4.3 从Web脚本连接MySQL服务器

我们在 8.3 节开发了一些命令行脚本，那些脚本在与 MySQL 服务器建立连接时使用的是同一段序文。大多数 CGI 脚本都共享一些序文代码，但还是有一些差异：

```
#!/usr/bin/perl

use strict;
use warnings;
use DBI;
use CGI qw(:standard);

use Cwd;
# option file that should contain connection parameters for UNIX
my $option_file = "/usr/local/apache/conf/sampdb.cnf";
my $option_drive_root;
```

```
# override file location for Windows
if ($^O =~ /^MSWin/i || $^O =~ /^dos/)
{
  $option_drive_root = "C:/";
  $option_file = "/Apache/conf/sampdb.cnf";
}

# construct data source and connect to server (under Windows, save
# current working directory first, change location to option file
# drive, connect, and then restore current directory)
my $orig_dir;
if (defined ($option_drive_root))
{
  $orig_dir = cwd ();
  chdir ($option_drive_root)
    or die "Cannot chdir to $option_drive_root: $!\n";
}
my $dsn = "DBI:mysql:sampdb;mysql_read_default_file=$option_file";
my %conn_attrs = (RaiseError => 1, PrintError => 0, AutoCommit => 1);
my $dbh = DBI->connect ($dsn, undef, undef, \%conn_attrs);
if (defined ($option_drive_root))
{
  chdir ($orig_dir)
    or die "Cannot chdir to $orig_dir: $!\n";
}
```

上面这段代码与我们在前面的命令行脚本里使用的同功能序文有以下几点差异。

- 上面这段代码的开头部分包含有 `use CGI` 和 `use Cwd` 两条语句。第一条语句用来导入 CGI.pm 模块。第二条语句所导入的模块将返回当前工作子目录的路径名，当脚本在 Windows 下运行时，就会使用这个语句。
- 上面这段代码不再解析来自命令行的连接参数，我们假设把这些参数列在一个选项文件里。
- 上面这段代码没有使用 `mysql_read_default_group` 选项来读取多个标准选项文件。我们将使用 `mysql_read_default_file` 选项来只读取一个用于 Web 脚本访问 `sampdb` 数据库而准备的选项文件。正如大家看到的那样，上面这段代码将从/usr/local/apache/conf/sampdb.cnf 文件（如果你使用的是 Unix 系统）或 C:\Apache\conf\sampdb.cnf 文件（如果你使用的是 Widows 系统）读取选项。在 Windows 系统上，上面这段代码会在连接之前先把当前路径切换到选项文件所在的硬盘根目录，等连接之后再切换回原来的子目录。至于为什么要做这种路径切换的理由，我们已经在 8.2.9 节里解释过了。

在 `sampdb` 发行版本里有一份现成的 `sampdb.cnf` 文件，你可以把它直接安装到系统里供基于 DBI 的 Web 脚本使用。它的内容如下所示：

```
[client]
host=localhost
user=sampadm
password=secret
```

如果你想在自己的系统上试用本章开发的基于 Web 的脚本，请把这个选项文件的位置修改为你打算使用的位置。你可能还需要安装 `sampdb.cnf` 选项文件，并在其中把有关参数修改为你将要使用的 MySQL 服务器主机名以及你的 MySQL 账户名和口令。

如果你使用的是 Unix 系统，那就还应该把这个选项文件的属主设置为你将用来运行 Apache 服务器的那个用户，并把这个文件的访问模式设置为 400 或 600 以阻止其他用户读取它。否则，在你的 Web 服务器主机上有登录账户的其他用户就能直接读取这个选项文件——这绝对是一个系统安全漏洞。

令人遗憾的是，即使你这样做了，那些有权在你的 Web 服务器上安装可执行脚本的其他用户仍能读取这个选项文件。我们知道，被 Web 服务器调用执行的脚本拥有用来运行 Web 服务器的那个登录账户的权限。也就是说，有权安装 Web 脚本的其他用户完全能够编写并执行一个这样的脚本，它将在运行时打开选项文件并把里面的内容显示在一个 Web 页面里。因为那个脚本拥有 Web 服务器用户的所有权限，所以它完全有权去读取你的选项文件，而你用来连接 MySQL 服务器和访问 `sampdb` 数据库的连接参数将暴露无遗。如果只有你能访问 Web 服务器主机，那么这无关紧要。如果你信不过你系统上的其他用户，就应该考虑采取这样一种对策：先创建一个在 `sampdb` 数据库上只有只读（`SELECT`）权限的 MySQL 账户，再把这个新建账户的用户名和口令（不是你本人的 MySQL 用户名和口令）列在 `sampdb.cnf` 选项文件里。这样就防止了有权修改你数据表的 MySQL 账户连接到你的数据库。创建一个权限受到限制的 MySQL 用户账户的办法请见第 12 章。这一策略的缺陷是：因为你使用的是一个只拥有只读权限的 MySQL 账户，所以你编写出来的脚本只能进行数据检索，而不能进行数据录入。

另一种办法是设法在 Apache 的 suEXEC 机制下执行脚本。其具体做法是：先以指定的可信任用户的身份运行脚本，再编写脚本从选项文件（只有那个用户能读取）里读取连接参数。

再有一种办法是让你的脚本要求由使用者提供一个 MySQL 用户名和口令，再用这个用户名和口令去连接 MySQL 服务器。这个办法比较适用于那些使用频率不高的系统管理类脚本，用在日常使用的脚本里往往会让人觉得比较麻烦。而且，万一有人在 Web 服务器与你的浏览器之间的网络上偷偷地放上了一个嗅探器，这种要求使用者提供用户名和口令的做法反而容易发生泄密，所以你很可能不得不建立一个安全连接，这些内容超出了本书的讨论范围。

从上面这些讨论不难得出一个结论：Web 脚本的安全性是一个非常复杂的问题。这个问题的涉及面是如此之广，所以你得自己加强这方面的学习，我很难再给出更好的建议了。在前面提到的 *MySQL and DBI for Web* 里有一章专门讲述网络安全，包括如何使用 SSL 来建立安全连接的步骤。Apache 服务器的使用手册里也有不少关于网络安全的讨论，下面这个网址上的网络安防 FAQ（常见问题答疑）也非常值得一读：

http：//www.w3.org/ Security/ Faq

8.4.4　一个基于Web的数据库浏览器

我们的第一个基于 Web 的 MySQL 应用程序是一个简单的脚本，`db_browse.pl`，它能帮你查出 sampdb 数据库里现有哪些数据表，并能让你从 Web 浏览器上交互地查看其中任何一个数据表的内容。这个脚本的工作情况是这样的。

- 当你从浏览器第一次请求 `db_browse.pl` 脚本时，它将连接到 MySQL 服务器，检索出 `sampdb` 数据库里的现有数据表并在一个页面里显示成链接（每个表一个链接）。当你在页面里点击某个数据表名字链接时，你的浏览器将向 Web 服务器发出一个请求，让 `db_browse.pl` 脚本显示该数据表的内容。
- 如果 `db_browse.pl` 脚本在它被调用时发现你选中了某个数据表的名字，它就会把这个数据表的内容检索出来并把这些信息显示在你的 Web 浏览器里。每列的头部就是数据表内数据列

的名字，显示为超链接，如果你点击了某个数据列的名字，浏览器就会向 Web 服务器请求重新显示该数据表内容，但这一次将按你选定的那个数据列对有关信息排序。

警告　在继续学习之前，我想先提醒大家一句：虽然 db_browse.pl 脚本体现了很多 Web 程序设计中的重要概念，但它本身却可能成为一个安全漏洞。这些脚本在 sampdb 中显示任何表，这可能是个麻烦。我们将在第 9 章编写一个能够让“美国历史研究会”会员用来通过 Web 编辑其会员记录的脚本，这个脚本对会员记录项的访问是由存放在 member_pass 数据表里的口令控制的。如果你到了那个时候还把 db_browse.pl 脚本可用，那么任何人都能查看这个口令数据表里的内容，也就能访问那些修改 member 数据表里的项所必需的信息。在你试用过脚本并了解了其工作原理后立刻把它移出 cgi-bin 子目录，这是个好主意。（另一种办法是把这个脚本安装在一个不可信任用户无法访问的私用 Web 服务器上。）

好了，如果你还没有被上面这段文字吓倒，就请和我一起看看 db_browse.pl 脚本的代码吧。这个脚本先生成 Web 页面的开头部分，然后检查 tbl_name 参数看是否需要显示某个数据表：

```
#!/usr/bin/perl
# db_browse.pl - Allow sampdb database browsing over the Web

use strict;
use warnings;
use DBI;
use CGI qw (:standard escapeHTML escape);

# ...set up connection to database (not shown)...

my $db_name = "sampdb";

# put out initial part of page
my $title = "$db_name Database Browser";
print header ();
print start_html (-title => $title, -bgcolor => "white");
print h1 ($title);

# parameters to look for in URL
my $tbl_name = param ("tbl_name");
my $sort_col = param ("sort_col");

# If $tbl_name has no value, display a clickable list of tables.
# Otherwise, display contents of the given table.  $sort_col, if
# set, indicates which column to sort by.

if (!defined ($tbl_name))
{
  display_table_names ($dbh, $db_name)
}
else
{
  display_table_contents ($dbh, $db_name, $tbl_name, $sort_col);
}
```

```
print end_html ();
```

要知道参数的值很容易，因为 CGI.pm 模块完全负责获取 Web 服务器传递给脚本的信息。我们只需把参数的名字送入 param()函数就可以了。db_browse.pl 脚本的主体中，参数名为 tbl_name，如果它没有值，就说明这是脚本的第一次调用，脚本就会生成一份数据表清单；如果它有值，脚本就会把 tbl_name 参数指定的数据表的内容按 sort_col 参数指定的数据列排序后显示出来。

display_table_names()函数负责生成初始页面。在 display_table_names()函数检索数据表清单并生成一个项目符号清单，其中每一项都是 sampdb 数据库里的数据表的名字：

```
sub display_table_names
{
my ($dbh, $db_name) = @_;

  print p ("Select a table by clicking on its name:");

  # retrieve reference to single-column array of table names
  my $sth = $dbh->prepare (qq{
              SELECT TABLE_NAME FROM INFORMATION_SCHEMA.TABLES
              WHERE TABLE_SCHEMA = ? ORDER BY TABLE_NAME
            });
  $sth->execute ($db_name);

  # Construct a bullet list using the ul() (unordered list) and
  # li() (list item) functions.  Each item is a hyperlink that
  # re-invokes the script to display a particular table.
  my @item;
  while (my ($tbl_name) = $sth->fetchrow_array ())
  {
    my $url = sprintf ("%s?tbl_name=%s", url (), escape ($tbl_name));
    my $link = a ({-href => $url}, escapeHTML ($tbl_name));
    push (@item, li ($link));
  }
  print ul (@item);
}
```

li()函数负责生成每个清单项两端的
和
标记，ul()函数负责生成一组清单项两端的
和
标记。清单里的每个数据表名都被设置为一个超链接，点击某个超链接将导致 Web 服务器再次调用 db_browse.pl 脚本去显示有关数据表的内容。display_table_names()函数生成的结果如下所示：

```
<ul>
<li><a href="/cgi-bin/db_browse.pl?tbl_name=absence">absence</a></li>
<li><a href="/cgi-bin/db_browse.pl?tbl_name=grade_event">grade_event</a></li>
<li><a href="/cgi-bin/db_browse.pl?tbl_name=member">member</a></li>
...
</ul>
```

如果 tbl_name 参数在 db_browse.pl 脚本被调用时有值，脚本就会把这个值和数据列的名字（脚本将按数据列名字对查询结果进行排序）传递给 display_table_contents()函数：

```
sub display_table_contents
{
```

```
my ($dbh, $db_name, $tbl_name, $sort_col) = @_;
my $sort_clause = "";
my @rows;
my @cells;

  # if sort column is specified, use it to sort the results
  if (defined ($sort_col))
  {
    $sort_clause = " ORDER BY " . $dbh->quote_identifier ($sort_col);
  }

  # present a link that returns user to table list page
  print p (a ({-href => url ()}, "Show Table List"));

  print p (strong ("Contents of $tbl_name table:"));

  my $sth = $dbh->prepare (
              "SELECT * FROM "
            . $dbh->quote_identifier ($db_name, $tbl_name)
            . "$sort_clause LIMIT 200"
            );
  $sth->execute ();

  # Use the names of the columns in the database table as the
  # headings in an HTML table.  Make each name a hyperlink that
  # causes the script to be reinvoked to redisplay the table,
  # sorted by the named column.

  foreach my $col_name (@{$sth->{NAME}})
  {
    my $url = sprintf ("%s?tbl_name=%s;sort_col=%s",
                       url (),
                       escape ($tbl_name),
                       escape ($col_name));
    my $link = a ({-href => $url}, escapeHTML ($col_name));
    push (@cells, th ($link));
  }
  push (@rows, Tr (@cells));

  # display table rows
  while (my @ary = $sth->fetchrow_array ())
  {
    @cells = ();
    foreach my $val (@ary)
    {
      # display value if non-empty, else display non-breaking space
      if (defined ($val) && $val ne "")
      {
        $val = escapeHTML ($val);
      }
      else
      {
        $val = " ";
      }
```

```
            push (@cells, td ($val));
          }
          push (@rows, Tr (@cells));
        }

        # display table with a border
        print table ({-border => "1"}, @rows);
    }
```

查询还使用了一条LIMIT 200子句，即把查询返回的数据行个数限制为200条，这是为避免脚本发送大量数据（sapmdb数据库里的数据表都没有这种情况，但如果你用这个脚本去查看其他数据库里的数据表内容，这个预防措施就能发挥作用了）到浏览器而采取的预防措施。display_table_contents()函数将把数据表里的数据行显示在一个HTML表格里：它用th()和td()函数生成HTML表格的表头和表格单元，用Tr()函数把表格单元组成表格行，再用table()函数生成行两端的<table>标记。

HTML表格各列的标题其实都是一些超链接，当你单击它们时，那个数据库表将重新显示。这些超链接都有一个sort_col参数，这个参数将明确指定数据列对数据表内容排序。比如说，在用来显示grate_event数据表内容的页面里，列标题所对应的超链接将是以下几个：

```
<a href="/cgi-bin/db_browse.pl?tbl_name=grade_event&sort_col=date">
date</a>
<a href="/cgi-bin/db_browse.pl?tbl_name=grade_event&sort_col=category">
category</a>
<a href="/cgi-bin/db_browse.pl?tbl_name=grade_event&sort_col=event_id">
event_id</a>
```

display_table_contents()函数里还使用了这样一个技巧：把空值转换为一个不间断空格( )。若有表框的HTML表格里有空值，有些浏览器就不能生成正确的表框边界线，把空表格值转换为一个不间断空格就能解决这一问题。

如果你想编写一个更通用的脚本，可以修改db_browse.pl脚本以使它能够浏览多个数据库。比如说，你可以增加一些代码，让它在服务器上显示一份数据库清单而不是只能列出特定数据库里的数据表清单。等你选择了某个数据库，就可获取数据表清单。

8.4.5　考试记分项目：考试分数浏览器

下一个脚本，score_browse.pl，用于显示考试记分项目记录的考分。严格地讲，我们得先录入考分再想方法检索它们。不过，下一章才会介绍录入考分的脚本，我们现在有几组记分阶段早期获得的分数。虽然我们还没有一个方便的分数录入手段，但我们可以编写脚本显示这些分数。这个脚本将把测验或考试分数显示为一个有序清单，这为确定评分曲线并给学生评定字母字母表示的考分级别的工作提供了方便。

score_browse.pl脚本与同为信息浏览器的db_browse.pl脚本有很多相似之处，但它的用途更具体——查看学生们的某次考试或测验分数。它生成的初始页面是一个考试事件清单，当你选中某次考试事件时，就可以查看到学生们在那次考试或测验中的分数了。每次事件的考试分数将按分数由高到低的顺序排序，你可以利用这个结果来确定评分曲线。

score_browse.pl脚本只需检查event_id这一个参数就能确定是否指定了事件。如果没有，score_browse.pl脚本将把grade_event数据表里的数据行显示出来供你选择；如果有，它就会显

示与被选中事件相关联的分数：

```
# ...set up connection to database (not shown)...

# put out initial part of page
my $title = "Grade-Keeping Project -- Score Browser";
print header ();
print start_html (-title => $title, -bgcolor => "white");
print h1 ($title);

# parameter that tells us which grade event to display scores for
my $event_id = param ("event_id");

# if $event_id has no value, display the event list.
# otherwise display the scores for the given event.
if (!defined ($event_id))
{
  display_events ($dbh)
}
else
{
  display_scores ($dbh, $event_id);
}

print end_html ();
```

display_events()函数负责提取 grade_event 数据表里的信息并把它们显示在一个表里，表格列的标题就是查询中给出的数据列的名字。在 HTML 表的每一行上，event_id 值将被设置为一个超链接，点击这个超链接，你就能看到学生们在这次考试中得到的分数。与各次考试事件相对应的 URL 地址由 score_browse.pl 脚本的路径再加上一个用来给出考试事件编号的参数构成，如下所示：

```
/cgi-bin/score_browse.pl?event_id=n
```

下面是 display_events()函数的代码：

```
sub display_events
{
my $dbh = shift;
my @rows;
my @cells;

  print p ("Select an event by clicking on its number:");

  # get list of events
  my $sth = $dbh->prepare (qq{
              SELECT event_id, date, category
              FROM grade_event
              ORDER BY event_id
            });
  $sth->execute ();

  # use column names for table column headings
  for (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
  {
```

```
    push (@cells, th (escapeHTML ($sth->{NAME}->[$i])));
  }
  push (@rows, Tr (@cells));

  # display information for each event as a row in a table
  while (my ($event_id, $date, $category) = $sth->fetchrow_array ())
  {
    @cells = ();
    # display event ID as a hyperlink that reinvokes the script
    # to show the event's scores
    my $url = sprintf ("%s?event_id=%d", url (), event_id;
    my $link = a ({-href => $url}, escapeHTML ($event_id));
    push (@cells, td ($link));
    # display event date and category
    push (@cells, td (escapeHTML ($date)));
    push (@cells, td (escapeHTML ($category)));
    push (@rows, Tr (@cells));
  }

  # display table with a border
  print table ({-border => "1"}, @rows);
}
```

当你选中某次考试事件时，浏览器将发送一条调用 score_browse.pl 脚本的请求，其末尾带有事件编号。score_browse.pl 脚本再通过 display_scores()函数把学生们在 event_id 参数所指定的那次考试中得到的分数显示出来。这个函数还会显示一个文本为“Show Events List”（显示考试事件清单）的超链接，点击这个超链接将使你返回初始页面，这样，你就能迅速返回到考试事件清单页面并作出下一次选择了。

```
sub display_scores
{
my ($dbh, $event_id) = @_;
my @rows;
my @cells;

  # Generate a link to the script that does not include any event_id
  # parameter.  If the user selects this link, the script will display
  # the event list.
  print p (a ({-href => url ()}, "Show Event List"));

  # select scores for the given event
  my $sth = $dbh->prepare (qq{
              SELECT
                student.name,
                grade_event.date,
                score.score,
                grade_event.category
              FROM
                student INNER JOIN score INNER JOIN grade_event
              ON
                student.student_id = score.student_id
                AND score.event_id = grade_event.event_id
              WHERE
```

```
                 grade_event.event_id = ?
               ORDER BY
                 grade_event.date ASC,
                 grade_event.category ASC,
                 score.score DESC
  });
  $sth->execute ($event_id);  # bind event ID to placeholder in query

  print p (strong ("Scores for grade event $event_id"));

  # use column names for table column headings
  for (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
  {
    push (@cells, th (escapeHTML ($sth->{NAME}->[$i])));
  }
  push (@rows, Tr (@cells));

  while (my @ary = $sth->fetchrow_array ())
  {
    @cells = ();
    foreach my $val (@ary)
    {
      # display value if non-empty, else display non-breaking space
      if (defined ($val) && $val ne "")
      {
        $val = escapeHTML ($val);
      }
      else
      {
        $val = " ";
      }
      push (@cells, td ($val));
    }
    push (@rows, Tr (@cells));
  }

  # display table with a border
  print table ({-border => "1"}, @rows);
}
```

display_scores()函数执行的数据库查询命令与 1.4.9 节中的第 10 小节里介绍如何编写联结时给出的一个语句很相似，在那节里，我们是根据一个给定的日期值来检索考分的，因为日期值要比抽象的 event_id 值更容易理解。可在这个 score_browse.pl 脚本里，我们却是根据 event_id 值来检索考分的。我们之所以会这样做，并不是因为我们是从 event_id 考虑的，而是因为这个脚本已经显示了一系列 ID 及其日期和考试类型供我们选择。说得更明白一点，这种操作接口使我们不必知道具体的细节就能完成我们想做的事情。我们用不着知道考试事件 ID，只要会识别事件的日期就行了，脚本会将它与正确的 ID 关联。

8.4.6 美国历史研究会：寻找志趣相同的会员

db_browse.pl 和 score_browse.pl 脚本通过一系列超链接在初始页面向用户提供一组选项，当

使用者作出选择时，被选中的超链接就将以预定的参数值再次调用这个脚本。Web 脚本向用户传递信息的另一种办法是在 Web 页面里提供一个供用户填选的表单，当选择范围不是容易确定的值时，这种办法无疑更为适用。下一个脚本就将使用这种机制来请求用户输入。

8.3 节曾编写过一个名为 interests.pl 的脚本来为“美国历史研究会”的会员查找有共同兴趣的其他会员。不过，会员不能访问那个脚本，所以就必须由研究会的秘书从命令行去执行这个脚本，然后再把结果寄给提出这种要求的会员。要是能把这种查找功能的可用范围扩大到每一位会员，那可就太好了，编写一个基于 Web 的脚本就能实现。本节将介绍两种数据表搜索技术：第一种以模式匹配为基础，第二种则使用了 MySQL 数据库提供的 FULLTEXT 搜索功能。

1. 利用模式匹配来进行搜索

我们将要编写的第一个搜索脚本叫做 ushl_browse.pl，它将提供一个表单供你输入一个关键字。当你提交这个表单时，服务器将再次调用这个脚本到 member 数据表里去搜索符合条件的会员并显示结果。这个脚本的搜索机制是这样的：先在你输入的关键字的两端分别加上一个“%”通配符，然后执行 LIKE 模式匹配把 interests 数据列里包含有这个关键字的行找出来。

这个脚本的主要工作是显示那个关键字表单，它还检查你是否提交了一个关键字，如果是，则执行搜索：

```
my $title = "U.S. Historical League Interest Search";
print header ();
print start_html (-title => $title, -bgcolor => "white");
print h1 ($title);

# parameter to look for
my $keyword = param ("keyword");

# Display a keyword entry form.  In addition, if $keyword is defined,
# search for and display a list of members who have that interest.

print start_form (-method => "post");
print p ("Enter a keyword to search for:");
print textfield (-name => "keyword", -value => "", -size => 40);
print submit (-name => "button", -value => "Search");
print end_form ();

# connect to server and run a search if a keyword was specified
if (defined ($keyword) && $keyword !~ /^\s*$/)
{
  # ... set up connection to database (not shown) ...
  search_members ($dbh, $keyword);
  # ... disconnect (not shown) ...
}
```

这个脚本向自己传递信息的办法与 db_browse.pl 或 score_browse.pl 脚本不同。这个脚本根本没有把参数追加到 URL 地址末尾。浏览器负责对表单里的信息编码，然后将其放在 POST 请求里发送。不过，不管信息以何种方式发送，CGI.pm 模块中的 param()函数都能返回参数，这是 CGI.pm 模块在简化 Web 程序设计工作方面的又一大贡献。

关键字的搜索由 search_members()函数完成。它以一个数据库句柄和你在 Web 页面里输入的那个关键字为输入参数，然后运行搜索查询并把找到的会员记录项显示出来：

```
sub search_members
{
my ($dbh, $interest) = @_;

  print p ("Search results for keyword: " . escapeHTML ($interest));
  my $sth = $dbh->prepare (qq{
                SELECT * FROM member WHERE interests LIKE ?
                ORDER BY last_name, first_name
              });
  # look for string anywhere in interest field
  $sth->execute ("%" . $interest . "%");
  my $count = 0;
  while (my $ref = $sth->fetchrow_hashref ())
  {
    html_format_entry ($ref);
    ++$count;
  }
  print p ("Number of matching entries: $count");
}
```

在运行 ushl_browse.pl 脚本时，你将发现你每次提交的关键字都会出现在下一个页面中的表单里，即使脚本在生成表单时把 keyword 字段的值设定为一个空字符串，也仍是如此。这是因为：如果在脚本的执行环境里存在某个字段的值，CGI.pm 模块就会将其自动填充到那个字段里。如果你不喜欢这种行为并想让这个字段每次都显示为空白，就需要给 textfield()调用增加一个 override 参数，如下所示：

```
print textfield (-name => "keyword",
                 -value => "",
                 -override => 1,
                 -size => 40);
```

search_members()函数使用了一个辅助函数 html_format_entry()来显示各个项。这个函数与之前为 gen_dir.pl 脚本编写的同名函数大同小异(请参见 8.3.1 节)。不过，这个函数用于 gen_dir.pl 脚本时是通过直接打印标记来生成 HTML 的，而用于 ushl_browse.pl 脚本时却是通过 CGI.pm 函数来生成标记的：

```
sub html_format_entry
{
my $entry_ref = shift;

  # encode characters that are special in HTML
  foreach my $key (keys (%{$entry_ref}))
  {
    next unless defined ($entry_ref->{$key});
    $entry_ref->{$key} = escapeHTML ($entry_ref->{$key});
  }
  print strong ("Name: " . format_name ($entry_ref)), br ();
  my $address = "";
  $address .= $entry_ref->{street}
                if defined ($entry_ref->{street});
  $address .= ", " . $entry_ref->{city}
                if defined ($entry_ref->{city});
  $address .= ", " . $entry_ref->{state}
```

```
                  if defined ($entry_ref->{state});
    $address .= " " . $entry_ref->{zip}
                  if defined ($entry_ref->{zip});
    print "Address: $address", br ()
                  if $address ne "";
    print "Telephone: $entry_ref->{phone}", br ()
                  if defined ($entry_ref->{phone});
    print "Email: $entry_ref->{email}", br ()
                  if defined ($entry_ref->{email});
    print "Interests: $entry_ref->{interests}", br ()
                  if defined ($entry_ref->{interests});
    print br ();
  }
```

html_format_entry()函数使用了format_name()函数来把first_name、last_name和suffix数据列的取值合并为一个完整的会员姓名。它与前面为gen_dir.pl脚本而编写的同名函数完全一样。

2. 利用FULLTEXT索引搜索

"美国历史研究会"的会员们往往有很多兴趣。在 member 数据表的 interests 列里，这些兴趣以逗号分隔，就像下面这样：

```
Revolutionary War,Spanish-American War,Colonial period,Gold rush,Lincoln
```

可不可以用 ushl_browse.pl 脚本去搜索与任何一个关键字匹配的行呢？答案是：可以，但非常有局限性。你可以在搜索表单里输入多个关键字，但要想找到匹配行，必须构建一个更复杂的查询来查找与每个关键字匹配的行。完成会员兴趣搜索任务的另一种更为灵活的方法是使用一个 FULLTEXT 索引。（请参见 2.15 节。）

进行 FULLTEXT 之前，必须确保 member 数据表是一个 MyISAM 数据表。如果你使用其他存储引擎创建 member 数据表，可以先用下面这条 ALTER TABLE 语句把它转换为 MyISAM 数据表：

```
ALTER TABLE member ENGINE = MyISAM;
```

再用下面这条语句为 member 数据表创建索引：

```
ALTER TABLE member ADD FULLTEXT (interests);
```

只有这样，才能用 interests 数据列进行 FULLTEXT 搜索。sampdb 发行版本里的 ushl_ft_browse.pl 脚本是在 ushl_browse.pl 脚本的基础上编写出来的，它们二者只在负责构造检索命令的 search_members()函数上有区别。下面是供 ushl_ft_browse.pl 脚本使用的 search_members()函数的代码：

```
sub search_members
{
my ($dbh, $interest) = @_;

  print p ("Search results for keyword: " . escapeHTML ($interest));
  my $sth = $dbh->prepare (qq{
              SELECT * FROM member WHERE MATCH(interests) AGAINST(?)
              ORDER BY last_name, first_name
            });
  # look for string anywhere in interest field
  $sth->execute ($interest);
  my $count = 0;
```

```
  while (my $ref = $sth->fetchrow_hashref ())
  {
    html_format_entry ($ref);
    ++$count;
  }
  print p ("Number of matching entries: $count");
}
```

这个版本的 search_members()函数对上一节里的同名函数作了以下几处修改。

- 查询命令使用的是 MATCH()...AGAINST()子句而不是 LIKE 子句。
- 没有在关键字两端加上“%”通配符来把它转换为一个匹配模板。

完成上述改动之后，你可以通过 Web 浏览器调用 ushl_ft_browse.pl 脚本并在搜索表单里输入多个关键字（加不加逗号分隔符都不要紧）。这个脚本将把与你输入的关键字中的任何一个相匹配的会员记录查找出来。

这个脚本有很多地方值得改进。比如说，FULLTEXT 搜索可以同时对多个数据列进行搜索。只需先创建一个同时涵盖多个数据列的索引，再对 ushl_ft_browse.pl 脚本作相应的修改来搜索它们就能享受到这个好处。例如，你可以删除初始的 FULLTEXT 索引，再创建出一个同时涵盖 last_name 和 full_name 两个数据列的 FULLTEXT 索引：

```
ALTER TABLE member DROP INDEX interests;
ALTER TABLE member ADD FULLTEXT (interests,last_name,first_name);
```

接着，再把 search_members()函数里的 SELECT 语句从原来的 MATCH (interests)修改为 MATCH (interests、last_name、first_name)，就可以使用新索引了。

还可以进一步改进 ushl_ft_browse.pl 脚本：在表单里增加两个按钮，让用户在“匹配任何一个关键字”模式和“匹配全部关键字”模式之间作出选择。前一种正是现在的脚本使用的，后一种模式的实现需要修改语句，让它使用一个 IN BOOLEAN MODE 类型的 FULLTEXT 搜索，再在每个关键字的前面加上一个加号以表明它必须在搜索过程中得到匹配。有关布尔模式搜索的详细讨论请见 2.15.2 节。

第 9 章

用 PHP 编写 MySQL 程序

PHP 是一种用于编写包含内嵌代码的 Web 页面的脚本语言。当 Web 页面被访问时，内嵌代码就会被执行并生成动态内容，而这些内容将作为 Web 页面的一部分被送往客户的 Web 浏览器。本章将介绍如何编写基于 PHP 的 Web 应用程序来访问 MySQL。本书的第 6 章比较了 PHP 语言、C 语言、Perl DBI 模块等几种 MySQL API。

第 1 章创建了一个示例数据库 sampdb，并在其中为考试记分项目和美国历史研究会分别创建了一些数据表。本章将以这个数据库里的数据表为例展开讨论。这里的应用程序应该运行在 PHP 5 或更高的版本。

讨论本章的前提是 PHP 脚本将与 Apache Web 服务器配合使用，当然，你可以替换为其他服务器。另外，为了让 PHP 知道如何访问 MySQL 数据库，必须在构建 PHP 的过程中把 MySQL C 客户端库也链接上。（这一要求会在 mysqlnd 可用时提出来。）如果你需要上面提到的这几个软件中的任何一个，请参阅附录 A，从中还可知道如何获得本章各示例脚本代码，它们是 sampdb 发行版本的组成部分，如果下载了 sampdb 发行版本，你就用不着自己动手输入脚本了。与本章有关的脚本都放在 sampdb 发行版本的 phpapi 目录里。

在 Unix 系统上，PHP 既可以用作 Apache 服务器的一个模块，也可以像传统的 CGI 程序那样用作独立的解释器。在 Windows 系统上，PHP 只能被当做一个独立运行的程序，除非你使用的是 Apache 2.x。如果是后一种情况，你可以选择在运行 PHP 时把它当做 Apache 服务器的一个模块。在其他的平台上，应该尽可能把 PHP 当做模块运行，因为这有助于获得更好的性能。

PHP 与 MySQL 之间的编程接口主要有以下几种。

- 在 PHP 5 及更高的版本里，可以使用号称“MySQL 加强版”的 mysqli 扩展模块。这个扩展模块提供了两种调用样式。可以把它当做一组以 mysqli_*xxx*() 方式命名的函数来使用，也可以通过一个面向对象的接口来使用它。
- mysql 扩展模块是 PHP 与 MySQL 之间的传统编程接口。它由一系列以 mysqli_*xxx*() 方式命名的函数构成。这些函数当中的绝大多数都直接对应于同名的 MySQL C API 函数。这个扩展模块没有提供面向对象的接口。mysql 模块在能力上逊色于 mysqli 模块，这不仅是因为它在调用样式方面缺乏灵活性，更是因为它没有提供对 MySQL 4.1 及更高版本里的新增功能的访问支持。mysql 模块是在 MySQL 4.1 系列问世之前开发的，现在已经有些落伍了。
- PHP 与 MySQL 之间的其他编程接口并非只能用于 MySQL，或者说它们与特定数据库引擎的关系不那么紧密。我们将在这一章里使用的 PHP Data Object（PDO，PHP 数据对象）就是其中之一。这个扩展模块提供了一套与特定面向对象数据库无关的编程接口，其设计理念与 Perl

DBI 模块的很相似。PDO 扩展模块的体系架构由两个层次构成，顶层是一组固定不变的编程接口，底层则由一些针对不同数据库引擎的驱动程序构成。如果想从某个驱动程序切换到另一个，只要修改传递给连接调用的参数让它们适用于你打算使用的驱动程序即可。

PDO 只能在 PHP 5.0 或更高版本里使用，因为早期的 PHP 版本不支持它所使用的面向对象功能。

按照它们最初的设计思路，`mysql` 和 `mysqli` 扩展模块以及针对 MySQL 的 PDO 驱动程序都要与 MySQL C 客户端库（`libmysqlclient` 库，详见附录 G）进行链接。这就导致了这样一种后果：如果安装 PHP 的目的是为了访问 MySQL 数据库，PHP 就必定会与 MySQL 发行版本的某一部分形成依赖关系。目前仍在开发中的 `mysqlnd` 库将是 `libmysqlclient` 库的替代品。`mysqlnd` 库是一个基础函数库，它实现了同样的通信协议，但不依赖于 MySQL 发行版本的任何一部分。`mysqlnd` 库将从 PHP 5.3 版开始被纳入到 PHP 当中，而这意味着无需安装 MySQL 客户端库也将可以从 PHP 去访问 MySQL 数据库。

在这一章里，我们将只对讨论内容所涉及的 PDO 对象和方法进行介绍。对 PDO 的讨论则只限于与 MySQL 驱动程序有关的内容。针对其他数据库引擎的驱动程序市面上已有不少，但这里就不对它们进行讨论了。在附录 I 里可以查到一份比较详细 PDO 接口清单。也可以求教于 PHP 使用手册，它对 PHP 的所有功能都有描述。这份手册可以从 PHP 官方网站 http：//www.php.net/获得。

PHP 脚本的文件名通常都有一个统一的后缀，因为 Web 服务器必须根据后缀来识别 PHP 脚本并调用 PHP 解释器来执行它们。如果你使用的后缀不能被 Web 服务器识别，Web 服务器就会把 PHP 脚本当做普通文本。本章里的脚本将统一使用“.php”作为后缀扩展名。配置 Apache 服务器以使之识别你想使用的扩展名的具体步骤可以在附录 A 中查到。（如果你没有配置 Apache 服务器的权限，请向系统管理员询问正确的扩展名。）附录 A 还描述了如何设置 Apache 服务器，让它把任何一个名为 `index.php` 的脚本用作该脚本所在的那个子目录的默认主页，就像 Apache 服务器对待 `index.html` 文件那样。

若想试用本章编写的脚本，必须把它们安装到一个 Web 服务器能够访问的地方。本章采用的办法是把美国历史研究会项目和考试记分项目的各脚本分别放到 Apache 文档树顶级目录下的子目录 ushl 和 gp 里。如果想把 Web 服务器也设置成这样，那么现在就该创建这两个子目录。假设服务器运行在本地主机上，这两个子目录里的页面就将有如下所示的 URL 地址：

```
http://localhost/ushl/...
http://localhost/gp/...
```

比如说，如果将每个子目录里的主页命名为 `index.php`，就可以像下面这样去访问它们：

```
http://localhost/ushl/index.php
http://localhost/gp/index.php
```

如果已经把 Apache 服务器配置成使用 `index.php` 脚本作为子目录的默认页面，下面两个 URL 与上面两个 URL 将是等价的：

```
http://localhost/ushl/
http://localhost/gp/
```

要记得修改本章中的示例 URL，指向你自己的 Web 服务器，而不是 `localhost`。

9.1 PHP 概述

PHP 语言的基本用途是解释脚本以生成一个将被送往某个客户的 Web 页面，而那个脚本里通常夹杂着 HTML 文本与可执行代码。HTML 文本将按原样发送给客户，而其中的 PHP 代码会被执行并替

换为它生成的输出内容。因此，客户不会看到代码，只能看到这些代码生成的 HTML 页面。（本章里的 PHP 脚本所生成的页面都符合 XHTML 规范，而不是仅符合 HTML 规范。8.4.2 节的第 2 小节简要介绍了 XHTML。）

当 PHP 开始读取一个文件时，它只是把它遇到的东西复制到输出，就好像这个文件的内容完全是 HTML 那样的文字文本。当 PHP 解释器遇到一个特殊的起始标记时，它会从文本复制模式切换到 PHP 代码模式，并把该文件解释为可执行代码。当 PHP 解释器遇到另一个特殊的结束标记时，它又将从代码模式切换回文本模式。这意味着你可以把静态文本（HTML 部分）与动态生成的结果（PHP 代码部分的执行结果）混合，生成一个会随着这个脚本的调用环境而变化的 Web 页面。比如说，可以让 PHP 脚本来处理用户已在其中输入数据库搜索参数的表单。用户输入不同，每次提交表单时的搜索参数也就不同，于是脚本生成并返回给用户的搜索结果页面也将不同。

我们就从下面这个最简单的例子来开始 PHP 学习之旅吧：

```
<html>
<body>
<p>hello,world</p>
</body>
</html>
```

这个脚本实在是太简单了，它里面根本就没有 PHP 代码！你肯定会问："这么简单的脚本能有什么用？"问得好，答案是：搭建包含你要生成的页面的脚本，再往里面填充 PHP 代码，这往往是一个很不错的思路。而且，这种做法是绝对允许的，PHP 解释器绝不会有什么抱怨。

如果想把 PHP 代码添加到脚本里，就必须把它们放在特殊的起始标记"<?php"和结束标记"?>"之间，区别于周围的文本。在遇到起始标记"<?php"时，PHP 解释器就会从文本模式切换到 PHP 代码模式，并从那里开始把该文件的后续内容解释为可执行代码，直到它遇到结束标记"?>"。在最终生成的 Web 页面里，这两个标记之间的代码将被替换为执行它们而生成的输出。现在，修改一下上面那个例子，给它加上一小段 PHP 代码，如下所示：

```
<html>
<body>
<p><?php print ("hello, world"); ?></p>
</body>
</html>
```

这个脚本里的代码部分是最少的，只有一行。这行 PHP 代码的执行结果是打印出"hello, world"，这个结果将作为输出内容的一部分被送往客户的浏览器。因此，这个脚本所生成的 Web 页面与前面那个完全由 HTML 文本构成的脚本所生成的 Web 页面是等同的。

PHP 代码可以用来生成 Web 页面的任何一个部分。一个极端是整个脚本完全由 HTML 文本构成而不包含任何 PHP 代码，这我们刚才已经见识过了；另一个极端则是全部 HTML 内容都由代码生成，如下所示：

```
<?php
print ("<html>\n");
print ("<body>\n");
print ("<p>hello, world</p>\n");
print ("</body>\n");
print ("</html>\n");
?>
```

这 3 个例子证明了这样一件事：PHP 能够让你以非常灵活的方式生成 Web 输出，HTML 文本与 PHP 代码的“混合比例”完全由你自己来控制。PHP 的灵活性还表现在它并不要求 PHP 代码全都出现在同一个地方。你可以在 HTML 模式与 PHP 代码模式之间随意切换，想切换多少次都行。

除本章几个例子里使用的<?php 和?>以外，PHP 还允许使用其他风格的标记。PHP 支持使用的各种标记以及它们的用法可以在附录 I（需上网查阅）里查到。

可独立执行的 PHP 脚本

本章里的示例脚本都是为了供 Web 服务器调用以生成 Web 页面。不过，如果你有一个可独立运行的 PHP 版本，就可以用它在命令行上执行 PHP 脚本。比如说，假设你有一个如下所示的 hello.php 脚本：

```
<?php print ("hello, world\n" ); ?>
```

那么，可以用下面这条命令亲自在命令行上执行它：

```
% php hello.php
hello, world
```

这对脚本的开发工作是有好处的，因为你能够立刻发现脚本是否有语法错误或其他问题，而不必每修改一次脚本都通过浏览器去查看它的执行情况。出于这个考虑，即使你通常把 PHP 用作 Apache 服务器的一个模块，我也建议你再建立一个可独立运行的 PHP 版本。

9.1.1 一个简单的PHP脚本

如果 PHP 的用途只不过是以打印语句来生成一些用静态 HTML 文本也能实现的 Web 页面，它也就没多大用处了。PHP 的威力是它能够让脚本根据不同的调用动态生成不同的网页输出。本小节描述的脚本给出了演示这种能力的一个简单例子。虽然它相对短小，但比前面那几个例子却多了一些实质内容。它能让我们看到从 PHP 访问 MySQL 数据库和把结果用在一个 Web 页面里都很容易。这个脚本构成了美国历史研究会网站主页的简单的基本框架。随着学习的深入，我们将使这个脚本进一步完善，但它眼下还只能显示一条简短的欢迎消息和现有会员总人数：


```
<html>
<head>
<title>U.S. Historical League</title>
</head>
<body bgcolor="white">
<p>Welcome to the U.S. Historical League Web Site.</p>
<?php
# USHL home page

try
{
  $dbh = new PDO("mysql:host=localhost;dbname=sampdb", "sampadm", "secret");
  $dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  $sth = $dbh->query ("SELECT COUNT(*) FROM member");
  $count = $sth->fetchColumn (0);
  print ("<p>The League currently has $count members.</p>");
  $dbh = NULL;  # close connection
```

```
}
catch (PDOException $e) { } # empty handler (catch but ignore errors)
?>
</body>
</html>
```

欢迎消息是静态文本，所以用 HTML 文本最容易实现。会员总人数却是一项动态内容，会随着时间的推移发生变化，只能通过在执行时查询 sampdb 数据库里的 member 数据表来确定。为此，起始标记和结束标记里的代码完成了以下几项工作。

(1) 连接 MySQL 服务器，并把 sampdb 数据库设置为默认数据库。

(2) 为后续的 PDO 调用启用异常捕获机制，这使我们可以轻而易举地捕获错误而不必显式地测试它们。

(3) 向 MySQL 服务器发送一个用来确定美国历史研究会现有会员总人数的查询（具体做法是统计 member 数据表的数据行总数）。

(4) 根据查询结果生成了一条报告会员总人数的消息。

(5) 关闭与 MySQL 服务器的连接。

在 sampdb 发行版本的 phpapi/ushl 子目录里能找到这个脚本，它的文件名是 index.php。在对连接参数作了必要的修改并把这个 index.php 文件复制到 Web 服务器文档树的 ushl 子目录里之后，你就可以通过下面两个 URL 中的任何一个（请根据 Web 服务器的具体设置对主机名和路径名做相应的修改）来从浏览器访问这个脚本了：

```
http://localhost/ushl/
http://localhost/ushl/index.php
```

下面，对这个脚本下分解说明。第一步是连接 MySQL 服务器：

```
$dbh = new PDO("mysql:host=localhost;dbname=sampdb", "sampadm", "secret");
```

这里使用了 new PDO()语法来调用 PDO 类的构造函数。这个构造函数将尝试连接数据库服务器。如果尝试失败，它将抛出一个异常；否则，它将返回一个 PDO 对象作为一个数据库句柄。

new PDO()的第一个参数是一个被称为“数据源名”的字符串，它的第二和第三个参数用来连接服务器的用户名“sampdb”和口令“secret”。DSN 字符串告诉 PDO 应该使用哪一个驱动程序，紧跟着给出了该驱动程序专用的连接参数。对于 MySQL，驱动程序的名字是“mysql”，连接参数是运行着服务器的主机名和将被选为默认数据库的数据库的名字。这里使用的 DSN 字符串表明，运行 MySQL 服务器的主机是“localhost”，默认数据库的名字是“sampdb”。这两个跟在冒号后面给出的参数都是可选的。host 参数的默认值是“localhost”，所以这个参数实际上是可以省略的。如果你省略了 dbname 参数，则表明没有选定任何默认数据库。（DSN 字符串还有其他格式，也允许使用其他参数。详见网上资源附录 I。）

你也许正在担心把用户名和口令嵌在脚本里会让别人看到。这种担心是应该的。在正常情况下，它们不会出现在发送到客户的结果 Web 页面里，因为脚本内容将被输出所替代。可万一 Web 服务器因配置不当而没能识别出这个脚本需要它调用 PHP 来处理，它就会把脚本当做普通文本，连接参数就将暴露在别人面前。我们将在 9.1.2 节解决这个问题。

new PDO()调用返回的数据库句柄可以用来对 MySQL 数据库进行各种操作（如发送一条 SQL 语句让它执行）。在连接成功之后，PDO 调用将使用这样一种默认的出错处理模式：在发生错误时默默地退出执行，不报告任何出错消息，这意味着你必须明确地检查每个错误。为了简化出错处理环节，

这个脚本启用了针对 PDO 调用的异常捕获机制：

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

启用异常捕获机制之后，可以用一个 try/catch 语句块把错误“引导”到一个异常处理例程，而不必明确测试。如果没有使用 try/catch 语句块，异常处理机制将在发生错误时结束脚本。

接下来，这个示例脚本将调用数据库句柄的 query()方法，把会员人数查询命令发送到服务器，然后取回并显示这次查询的结果：

```
$sth = $dbh->query ("SELECT COUNT(*) FROM member");
$count = $sth->fetchColumn (0);
print ("<p>The League currently has $count members.</p>");
```

query()方法负责把查询命令发送到服务器去执行。请注意，在查询语句字符串的末尾不需要使用分号字符“;”或字符序列“\g”或“\G”作为结束标志，这与我们在 mysql 客户程序里发出语句时的做法是不一样的。query()方法用来发送将返回一些数据行的语句。（对现有数据行进行修改的语句需要使用另一个方法 exec()来发送。）query()方法将返回一个 PDOStatement 对象，该对象是一个用来处理结果集的语句句柄。

具体到这里的查询，它的结果集仅有一个数据行，而这个数据行又仅有一个数据列——一个代表会员总人数的计数值。脚本通过调用$sth 对象的 fetchColumn()方法取回了那个唯一的数据行并提取出了第一个数据列（第 0 号数据列）的值。

在输出计数值之后，脚本通过把数据库句柄设置为 NULL 关闭了与服务器的连接。这一步是可选的。如果没有关闭与服务器的连接，PHP 将在脚本执行结束后自动地关闭它。

在这个示例脚本里，与 MySQL 交互的代码都集中在一个 try 语句块里，这样就可以用一个相应的 catch 语句块来捕获和处理那些代码在发生错误时抛出的异常。比如说，连接失败时会自动抛出一个异常，setAttribute()调用将使随后的 PDO 调用也在失败时抛出一个异常。不过，这个示例里的 catch 语句块是空的，其效果是捕获并忽略各种错误。这意味着即使真的发生错误，这个示例脚本也不会报告任何消息，相应的 Web 主页上将只有欢迎消息，没有会员人数或与之相关的出错消息。（具体到这个例子，向正在访问 Web 网站的用户显示一条出错消息反而会让人感到困惑。）9.1.8 节讨论了出错处理的其他办法。

PHP 脚本里的变量

PHP 脚本里的变量不需要你事先声明，你开始使用某个变量的事实就是它存在的理由。我们的主页脚本使用了$dbh、$sth 和$count 3 个变量，都没有声明过。（有些上下文需要声明，比如在一个异常处理程序里或者当你需要在某个函数里引用一个全局变量的时候。）

不管将被用来表示哪种类型的值，变量名都必须用以一个美元字符（$）开头的标识符表示。但如果需要通过数组或者对象的变量名来访问该变量值里的各个元素，那就还得再额外增加点东西。如果变量$x 表示的是一个标量值（比如一个整数或者一个字符串），就可以直接通过变量名$x 访问；如果$x 表示的是一个以数字为下标的数组，就需要通过$x[0]、$x[1]等来访问它的元素；如果$x 表示的是一个以"yellow"或"large"之类的字符串为下标的数组（即所谓的关联数组），就需要通过$x["yellow"]或$x["large"]之类的构造来访问它的元素。PHP 数组甚至允许混合使用数字和关联元素。比如说，$x[1]和$x["large"]允许是同一个数组里的元素。如果$x 表示的是一

个对象，就需要通过`$x->property_name`之类的构造（例如`$x->yellow`和`$x->large`等）来访问该对象的属性。注意：PHP 不允许以数字作为对象的属性名，除非把数字用花括号包起来；所以`$x->{1}`是 PHP 里的一个合法构造，`$x->1`则不是。如果需要引用的属性名包含空格或其他非法字符，也可以用花括号把它括起来。

9.1.2　利用PHP库文件实现代码封装

PHP 脚本与 DBI 脚本的不同之处在于 PHP 脚本存放在 Web 服务器的文档树内，而 DBI 脚本通常位于 Web 文档树外的 cgi-bin 里。这就带来了一个安全问题：服务器的配置错误会导致 Web 文档树里的页面以普通文本的形式被发送到客户去。这意味着 PHP 脚本里用来连接 MySQL 服务器的用户名和口令被泄露到外界的风险要比 DBI 脚本里的更大。

我们最早为历史研究会主页编写的脚本就存在着这个问题，因为它里面的 MySQL 用户名和口令都是明文。我们现在要利用 PHP 所提供的两项功能把这些连接参数转移到脚本以外的地方：函数和头文件。我们将编写一个名为`sampdb_connect()`的函数来完成建立连接和返回数据库句柄的任务，再把这个函数保存到一个不是主脚本的一部分但可以在脚本里引用的库文件里去。这种库文件就是所谓的"包含文件"（include file）。这个办法有以下几个优点。

- **连接建立代码更容易编写。**我们只需要在`sampdb_connect()`帮助函数里写一遍连接参数就够了，用不着再在每一个需要连接服务器的脚本里都写一遍。把类似于这样的细节转移到脚本以外的库文件还有助于提高脚本的可读性，因为你可以把注意力集中在每个脚本独有的细节上而无需分散精力去编写通用的连接建立代码。
- **包含文件可以供多个脚本使用。**这将提高代码的可重复利用性，让它们更容易维护。全局性的改动只需在包含文件里进行一次，就可以体现在每一个引用了它们的脚本里。比如说，如果把`sampdb`数据库从`localhost`主机迁移到了`boa.snake.net`主机，只要在当初定义`sampdb_connect()`函数的那个包含文件里把`hostname`参数的值改过来就行了，用不着去修改一大堆的脚本。
- **我们可以把包含文件存放到 Apache 文档树以外的地方。**这意味着用户将无法直接从他们的浏览器请求包含文件，即使 Web 服务器的配置不当，它们的内容也不会泄露。如果不想让某些敏感的信息被 Web 服务器发送到网站以外的地方，利用包含文件来隐藏它们是个很好的策略。请注意，这可以改善系统的安全性是不假，但并不意味着用户名和口令就百分之百地安全了。如果没有采取其他一些必要的预防措施，在 Web 服务器主机上有登录账户（因而能够访问其文件系统）的其他用户将仍可以直接读取你的头文件。8.4.3 节给出了一些在安装 DBI 配置文件时需要注意的事项，就是为了保护它们免遭他人偷窥。类似的预防措施同样适用于 PHP 包含文件。

要想使用包含文件，必须先找个地方来存放它们，并把那个地点告诉 PHP。如果系统里已经有了一个这样的地点，你可以继续使用它。如果不是这样，请按以下步骤创建一个包含文件存放地点。

(1) 在 Web 服务器的文档树以外的某个地方创建一个子目录来存放 PHP 包含文件。在我自己的系统上，我为此而创建的子目录是/usr/local/apache/lib/php，它在我的文档树/usr/local/apache/htdocs 以外。

(2) 脚本可以通过包含文件的完整路径名去访问它们，若是已经设置好了 PHP 的搜索路径，只使用它们的基本名（路径名里的最后一项）也可以。后一种办法更方便一些，PHP 将替我们去寻找包含

文件。PHP 在寻找包含文件时使用的搜索路径由 PHP 的初始化文件 php.ini 里的 include_path 配置项控制。在你自己的系统上找到这个文件（我的安装在/usr/local/bin 子目录里），打开该文件找到 include_path 行。如果它现在没有任何值，把它设置为你刚刚为头文件而创建的那个子目录的完整路径名即可，如下所示：

```
include_path = "/usr/local/apache/lib/php"
```

如果 include_path 变量已经有了一个值，那就把新建子目录添加到那个值里：

```
include_path = "/usr/local/apache/lib/php:current_value"
```

在 Unix 系统上要像上面这样使用冒号字符来分隔 include_path 值里的子目录。在 Windows 系统上要使用分号来分隔它们。

修改完 php.ini 文件后，重新启动 Apache 以便让你的改动生效。

PHP 头文件的使用方法和 C 语言里的头文件很相似。比如说，PHP 在多个子目录里寻找 PHP 头文件的做法就和 C 预处理器在多个子目录里寻找 C 头文件的情况相类似。

(3) 创建你将使用的包含文件并存放到你为它们新建的子目录里去。这个文件应该有一个言简意赅的名字，我们将使用 sampdb_pdo.php。这个文件最终将包含好几个函数，但现在只有一个 sampdb_connect()函数：

```
<?php
# sampdb_pdo.php - common functions for sampdb PDO-based PHP scripts

# Function that uses our top-secret username and password to connect
# to the MySQL server to use the sampdb database. It also enables
# exceptions for errors that occur for subsequent PDO calls.
# Return value is the database handle produced by new PDO().

function sampdb_connect ()
{
  $dbh = new PDO("mysql:host=localhost;dbname=sampdb",
                 "sampadm", "secret");
  $dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
  return ($dbh);
}
?>
```

为了连接数据库服务器，sampdb_connect()函数构造了一个数据源的名字并把它连同 MySQL 账户的用户名和口令一起传递给了 new PDO()调用。接下来，它把出错处理模式设置成在发生 PDO 错误时抛出一个异常，然后返回一个数据库句柄供后续代码与服务器通信时使用。这个函数的用法如下：

```
$dbh = sampdb_connect ();
```

之所以要在 sampdb_connect()函数里激活异常捕获机制，是因为在库文件里启用它要比在每一个用到库文件的脚本里这样做更为方便。

请注意，sampdb_pdo.php 文件里的 PHP 代码出现在<?php 和?>脚本标签之间。这是因为 PHP 是在文本复制模式下开始读取包含文件的。如果省略了这些标签，PHP 会把头文件的内容当做普通文本发送出去而不是把它们解释为 PHP 代码。如果包含文件的内容就是你打算生成的 HTML 输出的话，这么做是可以的。但如果你想让它的内容得到执行，就必须把 PHP 代码用脚本标记括起来。

(4) 在从脚本里引用包含文件的具体办法有好几种，下面任何一条语句都可以使用：

```
include "sampdb_pdo.php";
require "sampdb_pdo.php";
include_once "sampdb_pdo.php";
require_once "sampdb_pdo.php";
```

PHP 对这 4 种语句的处理情况如下。

- include 和 require 语句都可以导入指定的文件并对其内容进行求值。它们两者的区别在于如果头文件没有找到，include 语句将生成一条警告消息并继续执行，而 require 语句将生成一条出错消息并退出执行。
- include_once 和 require_once 语句分别类似于 include 和 require 语句，但如果 PHP 已经读取过指定文件，将不会再次读取它。这在包含文件本身还包含着其他文件的情况下很有用，这有助于避免因为多次导入同一个头文件而引发“函数重复定义”错误。

本章中的脚本统一使用 require_once 语句。当 PHP 遇到文件导入语句的时候，它将寻找指定文件并读入它的内容，指定文件里的所有东西对接下来的脚本代码来说都是可访问的。

sampdb_pdo.php 文件可以在 sampdb 发行版本的 phpapi 子目录里找到。记得要把这个文件里的连接参数改成你在连接你自己的 MySQL 服务器时应该使用的值。把这个文件复制到你用来存放包含文件的子目录，设置好这个文件的访问权限和属主，让它只能由你的 Web 服务器来读取。

现在，我们来修改历史研究会的主页，让它引用 sampdb_pdo.php 包含文件并通过调用 sampdb_connect()函数的办法去连接 MySQL 服务器：

```
<html>
<head>
<title>U.S. Historical League</title>
</head>
<body bgcolor="white">
<p>Welcome to the U.S. Historical League Web Site.</p>
<?php
# USHL home page - version 2

require_once "sampdb_pdo.php";

try
{
  $dbh = sampdb_connect ();
  $sth = $dbh->query ("SELECT COUNT(*) FROM member");
  $count = $sth->fetchColumn (0);
  print ("<p>The League currently has $count members.</p>");
  $dbh = NULL;  # close connection
}
catch (PDOException $e) { } # empty handler (catch but ignore errors)
?>
</body>
</html>
```

如上所示的脚本代码已经收录在 sampdb 发行版本的 phpapi/ushl 子目录中的 index2.php 文件里。请把它复制到 Web 服务器文档树的 ushl 子目录，并重新命名为 index.php 以替换现有的同名文件。这个操作用一个更安全的版本替换掉了一个不太安全的版本，因为新文件里没有明文形式的 MySQL 用户名和口令。

有些读者可能已经注意到了，虽然使用了一个包含文件，但这个主页里的代码并没有明显的减少。别着急。sampdb_pdo.php 文件还可以容纳其他的函数，我们可以把它当做一个方便的工具箱，把我们在许多脚本里要用到的例程都放进去。事实上，我们现在就可以创建两个新函数放到这个文件里去。我们在本章后续内容里将要编写的每一个 Web 脚本都需要在页面的开头生成一组基本相同的 HTML 标记，在页面的结尾部分还要再生成一组。既然如此，与其在每个脚本里重复写出同样的 HTML 标记，不如编写一个 html_begin()和一个 html_end()函数来替我们完成。html_begin()函数需要几个参数来设定页面的窗口标题和文档标题。下面是这两个函数的代码：

```
function html_begin ($title, $header)
{
  print ("<html>\n");
  print ("<head>\n");
  if ($title != "")
    print ("<title>$title</title>\n");
  print ("</head>\n");
  print ("<body bgcolor=\"white\">\n");
  if ($header != "")
    print ("<h2>$header</h2>\n");
}

function html_end ()
{
  print ("</body>\n");
  print ("</html>\n");
}
```

把 html_begin()和 html_end()函数放到 sampdb_pdo.php 文件里以后，就可以修改历史研究会的主页来使用它们了。下面是改写后的脚本（index3.php）：

```
<?php
# USHL home page - version 3

require_once "sampdb_pdo.php";

$title = "U.S. Historical League";
html_begin ($title, $title);
?>

<p>Welcome to the U.S. Historical League Web Site.</p>

<?php
try
{
  $dbh = sampdb_connect ();
  $sth = $dbh->query ("SELECT COUNT(*) FROM member");
  $count = $sth->fetchColumn (0);
  print ("<p>The League currently has $count members.</p>");
  $dbh = NULL;  # close connection
}
catch (PDOException $e) { } # empty handler (catch but ignore errors)

html_end ();
?>
```

请注意，PHP 代码被分成了两个部分，出现在它们中间的是网站欢迎辞的 HTML 文本。请把 index3.php 复制到你 Web 服务器文档树的 ushl 子目录，并重新命名为 index.php 以替换现有的同名文件。

利用函数来生成页面的开头和结尾其实是一项很重要的功能。如果想把页头或页脚的视觉效果改成另一种风格，只需修改相关的函数，就可以让所有使用了它们的脚本都受到影响。比如说，如果想在历史研究会的每个页面的底部加上一条“Copyright USHL”消息，只要在 html_end()或其他类似用途的函数里加上这条消息就大功告成了。

9.1.3　简单的数据检索页面

嵌在美国历史研究会主页里的脚本所运行的查询只返回了一个数据行（会员总人数）。下一个脚本将演示如何处理由多个数据行构成的结果集（member 数据表的全部内容）。这个 PHP 脚本与在 8.2.2 节开发的 DBI 脚本 dump_members.pl 等效，所以给它起名为 dump_members.php。DBI 脚本通常是在命令行上被执行的，而 PHP 脚本通常要通过 Web 服务器来执行。因此，PHP 脚本的输出通常是 HTML 内容而不是以制表符分隔的文本。为了把数据行和数据列整齐地显示出来，dump_members.php 脚本将把会员记录写到一个 HTML 表格中。下面就是这个脚本的代码：

```
<?php
# dump_members.php - dump U.S. Historical League membership as HTML table

require_once "sampdb_pdo.php";

$title = "U.S. Historical League Member List";
html_begin ($title, $title);

$dbh = sampdb_connect ();

# issue statement
$stmt = "SELECT last_name, first_name, suffix, email,"
      . " street, city, state, zip, phone FROM member ORDER BY last_name";
$sth = $dbh->query ($stmt);

print ("<table>\n");          # begin table
# read results of statement, and then clean up
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  print ("<tr>\n");           # begin table row
  for ($i = 0; $i < $sth->columnCount (); $i++)
  {
    # escape any special characters and print table cell
    print ("<td>" . htmlspecialchars ($row[$i]) . "</td>\n");
  }
  print ("</tr>\n");          # end table row
}
print ("</table>\n");         # end table

$dbh = NULL;  # close connection

html_end ();
?>
```

sampdb_connect()函数启用了针对 PDO 错误的异常处理机制，但 dump_members.php 脚本没有包含任何用来处理异常的 try/catch 语句块。那么，如果出现错误会发生什么事情？具体到这个例子，PHP 的默认行为是结束脚本的执行并输出一条问题描述消息。这与我们在美国历史研究会主页脚本里使用的办法不同，我们在那里使用了一个内容为空的异常处理例程来捕获并忽略所有的错误。这是因为，那个主页脚本的主要功用是显示一条欢迎信息，显示现有会员总人数只是一项副业，所以不需要在检索不到总人数时打印出错信息。但 dump_members.php 脚本的存在价值就在于显示从数据库查询出来的结果，所以当它因为某种原因而无法显示数据库查询结果时，就应该给出一条出错信息以表明发生了什么样的错误。

在发出从 member 数据表选取数据行的查询命令之后，脚本调用了 fetch()方法，它将返回结果集里的下一个数据行；如果已经到达结果集的末尾，返回 FLASE。PDO::FETCH_NUM 参数的作用是让 fetch()方法把数据行返回为一个通过数值索引来访问的数组。

为了对将被显示在 Web 页面里的值进行编码，dump_members.php 脚本使用 htmlspecialchars()函数对 HTML 语言里的特殊字符（如“<”、“>”或“&”）进行了转义处理。（如果是对将出现在 URL 字符串里的值编码，需要使用 urlencode()函数。）这两个 PHP 编码函数的功能与 8.4.2 节的第 3 小节讨论的供 Perl 脚本使用的 CGI.pm 库方法 escapeHTML()和 escape()很相似。

如果想试用一下 dump_members.php 脚本，请先把它安装到 Web 服务器文档树的 ushl 子目录里，再从 Web 浏览器使用下面这个 URL 访问它：

```
http://localhost/ushl/dump_members.php
```

为了让人们知道 dump_members.php 脚本，我们还需要在美国历史研究会的主页脚本里添加一个指向它的链接。修改后的主页脚本 index4.php 如下所示：

```
<?php
# USHL home page - version 4

require_once "sampdb_pdo.php";

$title = "U.S. Historical League";
html_begin ($title, $title);
?>

<p>Welcome to the U.S. Historical League Web Site.</p>

<?php
try
{
  $dbh = sampdb_connect ();
  $sth = $dbh->query ("SELECT COUNT(*) FROM member");
  $count = $sth->fetchColumn (0);
  print ("<p>The League currently has $count members.</p>");
  $dbh = NULL;  # close connection
}
catch (PDOException $e) { } # empty handler (catch but ignore errors)
?>

<p>
You can view the directory of members <a href="dump_members.php">here</a>.
```

```
</p>

<?php
html_end ();
?>
```

对于早前的主页，我们还需要把 index4.php 文件复制到 Web 服务器的文档树里的 ush1 子目录，并重命名为 index.php 以替换掉现用的同名文件。

dump_members.php 脚本演示了如何利用 PHP 脚本来检索 MySQL 数据库里的信息并生成为 Web 页面的内容。如果你想得到更精细的输出效果，可以自行修改这个脚本。比如说，你可以把来自 email 数据列的值显示为一个超链接而不是静态文本，这样，站点访问者只需点击这个超链接就可以向研究会的会员发电子邮件了。sampdb 发行版本里的 dump_members2.php 脚本能够完成这一任务。它与 dump_members.php 脚本只在那个用来取回和显示会员项的循环语句里稍有区别。下面是 dump_members.php 脚本里的循环语句：

```
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  print ("<tr>\n");            # begin table row
  for ($i = 0; $i < $sth->columnCount (); $i++)
  {
    # escape any special characters and print table cell
    print ("<td>" . htmlspecialchars ($row[$i]) . "</td>\n");
  }
  print ("</tr>\n");           # end table row
}
```

会员们的电子邮件地址保存在查询结果的第四个数据列里。于是，如果第四个数据列里的值不为空，dump_members2.php 脚本就把它生成为一个超链接以区别于其他数据列。

```
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  print ("<tr>\n");            # begin table row
  for ($i = 0; $i < $sth->columnCount (); $i++)
  {
    print ("<td>");
    # escape any special characters and print table cell;
    # email is in column 4 (index 3) of result
    if ($i == 3 && $row[$i] != "")
    {
      printf ("<a href=\"mailto:%s\">%s</a>",
              $row[$i],
              htmlspecialchars ($row[$i]));
    }
    else
    {
      print (htmlspecialchars ($row[$i]));
    }
    print ("</td>\n");
  }
  print ("</tr>\n");           # end table row
}
```

9.1.4 处理语句结果

PDO 提供了以下几种执行 SQL 语句的手段。

- PDO 对象的 exec()和 query()方法都可以接受一条 SQL 语句作为参数，立刻执行该语句并返回结果。
  - 诸如 DELETE、INSERT、REPLACE 和 UPDATE 之类用来修改数据行的语句需要通过调用 exec()方法来执行，这个方法将返回一个计数值来表明那条语句实际改变（删除、插入、替换、刷新等）了多少个数据行。
  - 诸如 SELECT 之类会生成一个结果集的语句需要通过调用 query()方法来执行，这个方法将返回一个 PDOStatement 语句句柄对象。你可以通过这个对象进一步获得关于结果集的信息。比如说，如果想知道结果集里有多少个数据列，可以调用 columnCount()方法；如果你想访问结果集里的数据行，可以调用 fetch()方法。
- PDO 还支持通过预处理语句而实现的两阶段语句执行机制。PDO 对象的 prepare()方法接受一条 SQL 语句作为参数，但不会立刻执行那条语句，prepare()方法将进行一些初始化处理并返回一个 PDOStatement 语句句柄对象。这个语句句柄有一个 execute()方法用来执行该语句，还有其他一些方法用来处理结果集。

  prepare()和 execute()方法可以用来处理任何一种语句，而非仅适用于修改数据行的语句或返回数据行的语句。

  预处理语句机制还提供了两种重要的能力：一是可以反复多次地执行，改善了性能；二是可以处理数据值里的特殊字符。详见 9.1.6 节。

在接下来的几节里，我们将更详细地讨论 PDO 的语句执行能力，示例程序将假设已启用了针对错误的异常处理机制。

1. 处理修改数据行的语句

对数据行进行修改的语句需要通过数据库句柄的 exec()方法来执行。exec()将返回一个数据行计数值以表明有多少数据行受到了影响。假设你需要从 member 数据表里把第 149 号会员的记录删掉并想知道删除操作是否成功，下面的例子演示了如何确定该语句是否真的删除了数据行：

```
$count = $dbh->exec ("DELETE FROM member WHERE member_id = 149");
if ($count > 0)
  print ("Member 149 was deleted\n");
else
  print ("No record for member 149 was found\n");
```

2. 处理返回结果集的语句

会生成结果集的语句需要通过调用数据库句柄的 query()方法来执行。query()方法将返回一个 PDOStatement 语句句柄对象供你访问结果集。语句句柄有好几个非常有用的方法，如下所示。

- fetch()方法。返回结果集里的下一个数据行，如果已经到达结果集的末尾，则返回 FALASE。
- fetchColumn()方法。类似于 fetch()方法，但只返回每行的一个数据列。
- columnCount()方法。返回结果集里的数据列个数。（数据行的个数不能直接通过调用某个方法查知，必须先取回数据行再统计它们。）

在前面讨论 USHL 主页时给出的几个例子演示了如何只调用一次 fetchColumn 方法来取回一个值。如果你打算取回的结果集包含多个数据行，每个数据行又包含多个数据列，常用的办法是通过在

一个循环里反复调用 fetch()方法来取回那些结果。下面的例子演示了一种具体的做法。为了确定结果集里有多少个数据行，我们还在取回数据行的同时对它们进行了统计：

```
$sth = $dbh->query ("SELECT * FROM member");
# fetch each row in result set
$count = 0;
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  # print values in row, separated by commas
  for ($i = 0; $i < $sth->columnCount (); $i++)
    print ($row[$i] . ($i < $sth->columnCount () - 1 ? "," : "\n"));
  $count++;
}
printf ("Number of rows returned: %d\n", $count);
```

fetch()方法有一个参数可以用来确定返回哪种值。表 9-1 列出了一些常用的取回模式。

表9-1　fetch()方法的数据行取回模式

输入参数	返　回　值
PDO::FETCH_ASSOC	一个数组，其元素需要通过关联索引来访问
PDO::FETCH_NUM	一个数组，其元素需要通过数值索引来访问
PDO::FETCH_BOTH	一个数组，其元素通过关联索引或数值索引来访问均可
PDO::FETCH_OBJ	一个对象，其元素需要通过属性来访问

fetch()方法的参数是可选的，如果没有它，则使用默认模式。如果你没有改变过默认设置，它将是 PDO::FETCH_BOTH，其含义是 fetch()方法将把每个数据行返回为一个数组，该数组的元素既可以通过数据列名访问，也可以通过数值下标访问。

有两种办法可以在取回数据行之前改变默认的取回模式，其一是向 query()方法多传递一个参数，其二是调用语句句柄的 setFetchMode()方法。下面两组代码都将把取回模式设置为 PDO::FETCH_NUM，对随后进行的结果集检索操作产生影响：

```
$sth = $dbh->query ($stmt, PDO::FETCH_NUM);

$sth = $dbh->query ($stmt);
$sth->setFetchMode (PDO::FETCH_NUM);
```

如果取回模式是 PDO::FETCH_ASSOC，fetch()方法将把结果集里的下一个数据行返回为一个关联数组，其中元素的名字就是在查询命令里选取的数据列的名字。比如说，如果从 president 数据表检索 last_name 和 first_name 值，就需要像下面这样访问数据列：

```
$stmt = "SELECT last_name, first_name FROM president";
$sth = $dbh->query ($stmt);
while ($row = $sth->fetch (PDO::FETCH_ASSOC))
  printf ("%s %s\n", $row["first_name"], $row["last_name"]);
```

如果取回模式是 PDO::FETCH_NUM，fetch()方法将把结果集里的下一个数据行返回为一个数组，其中的元素需要通过一个从零开始的数值索引来访问。结果集里的数据列的个数可以通过调用语句句柄的 columnCount()方法来确定。下面这个简单的循环将取回数据行的值并以制表符分隔的格式打印出来：

```
$stmt = "SELECT * FROM president";
$sth = $dbh->query ($stmt);
while ($row = $sth->fetch (PDO::FETCH_NUM))
{
  for ($i = 0; $i < $sth->columnCount (); $i++)
    print ($row[$i] . ($i < $sth->columnCount () - 1 ? "\t" : "\n"));
}
```

如果取回模式是 PDO::FETCH_BOTH，fetch()方法将返回数组，该数组的元素既可以通过数值索引、也可以通过数据列的名字来访问。这相当于 PDO::FETCH_ASSOC 和 PDO::FETCH_NUM 的组合。

如果取回模式是 PDO::FETCH_OBJ，fetch()方法将把结果集里的下一个数据行返回为一个对象，这个对象的属性需要使用$row->col_name 语法来访问：

```
while ($row = $sth->fetch (PDO::FETCH_OBJ))
  printf ("%s %s\n", $row->first_name, $row->last_name);
```

如果查询命令里包含一个计算出来的数据列该怎么办？比如说，如果你发出的是一条如下所示的查询命令，它返回的值将是一个表达式的计算结果：

```
SELECT CONCAT(first_name, ' ', last_name) FROM president
```

对于这样的查询命令，以对象形式取回数据行往往是行不通的。被选中的数据列的名字就是表达式本身，这不是合法的属性名。不过，我们可以通过给数据列起一个别名来提供一个合法的名字。下面这条查询给数据列起了一个别名叫做 full_name：

```
SELECT CONCAT(first_name, ' ', last_name) AS full_name FROM president
```

现在可以把查询里的每个数据行取回为一个对象了利用别名可以使用$row->full_name 语法去访问那个数据列。

以上示例都使用了下面这种形式的数据行取回循环来把每个数据行赋值给$row 变量：

```
while ($row = $sth->fetch ([fetch_mode]))
  ... handle row ...
```

不过，还有其他办法可以取回数据行，其中一种是取回一个数组并把结果赋值给一个变量列表。比如说，下面这段代码将把 last_name 和 first_name 数据列赋值给$ln 和$fn 变量，并按名在前、姓在后的顺序打印出那些姓名：

```
$stmt = "SELECT last_name, first_name FROM president";
$sth = $dbh->query ($stmt);
while (list ($ln, $fn) = $sth->fetch (PDO::FETCH_NUM))
  printf ("%s %s\n", $fn, $ln);
```

变量的名字可以任意选择，只要合法就行，但它们在 list()列表里的顺序必须与查询命令里选取数据列的顺序相对应。

还可以把各个数据列值直接检索到 PHP 变量里。这需要先用 bindColumn()方法把结果集里的数据列与有关变量绑定在一起，再使用 PDO::FETCH_BOUND 取回模式来取回数据行。此时，如果尚未到达结果集的末尾，fetch()方法将返回 TRUE 并把它取回的数据行里的数据列分别赋值给绑定变量：

```
$stmt = "SELECT last_name, first_name FROM president";
$sth = $dbh->query ($stmt);
$sth->bindColumn (1, $ln);
$sth->bindColumn (2, $fn);
```

```
while ($sth->fetch (PDO::FETCH_BOUND))
  printf ("%s %s\n", $fn, $ln);
```

fetchAll()方法可以一次性地把数据行全都取回到一个数组里：

```
$rows = $sth->fetchAll ();
```

类似于 fetch()方法，fetchAll()方法也有它自己的默认取回模式，并且也接受一个显式的取回模式参数。

语句句柄本身也可以用作一个迭代器，而不必显式调用 fetch()方法：

```
foreach ($sth as $row)
  printf ("%s %s\n", $row["first_name"], $row["last_name"]);
```

默认的取回模式决定着数据行将如何被取回。

9.1.5　测试查询结果里的NULL值

PHP 把结果集里的 SQL 语言的 NULL 值表示为 PHP 语言的 NULL 值。我们可以用函数 isnull()来检查某个 SELECT 查询所返回的数据列里是否包含 NULL 值。请看下面这段用来选择和打印 member 数据表里的姓名和电子邮件地址的代码，当电子邮件地址是 NULL 值时，它将打印一条“No email address available”（没有电子邮件地址）信息：

```
$stmt = "SELECT last_name, first_name, email FROM member";
$sth = $dbh->query ($stmt);
while (list ($last_name, $first_name, $email) = $sth->fetch (PDO::FETCH_NUM))
{
  printf ("Name: %s %s, Email: ", $first_name, $last_name);
  if (is_null ($email))
    print ("No email address available");
  else
    print ($email);
  print ("\n");
}
```

还可以用“===”操作符将一个值与 PHP 的 NULL 常数比较（连续三个等号）来测试 SQL 语言的 NULL 值，如下所示：

```
if ($email === NULL)
  print ("No email address available");
else
  print ($email);
```

PHP 语言中的 NULL 值等价于一个未定义值，所以还可以用 isset()函数来测试 NULL 值：

```
if (!isset ($email))
  print ("No email address available");
else
  print ($email);
```

9.1.6　使用预处理语句

前面介绍的 exec()和 query()方法都是执行 SQL 语句并立刻返回其结果集。PDO 也可以把 SQL 语句的预处理和执行分两个步骤来完成。具体做法是：先使用数据库句柄的 prepare()方法获得一个

语句句柄，再使用语句句柄去执行语句。

```
$sth = $dbh->prepare ($stmt);
$sth->execute ();
```

在执行语句之后，如果它修改数据行，你可以调用 rowCount()方法去查出它实际影响到多少行：

```
$count = $sth->rowCount ();
```

如果语句返回了一些数据行，可以调用 fetch()、columnCount()等方法进行处理。如果想知道结果集里有多少个数据行，需要在取回它们的同时对它们统计。(rowCount()方法只适用于修改数据行的语句。)

预处理语句有以下一些重要的能力。

- 语句字符串可以包含占位符而不仅仅是文字数据值。在对语句进行预处理之后，只需在执行该语句之前把特定的数据值绑定到相应的占位符，就可以让 PDO 替我们完成对特殊字符和 NULL 值的转义和引用。把值绑定到占位符的办法有好几种，详见 9.1.7 节。
- 经过预处理的语句可以反复多次地执行，而不用每次执行都需要预处理。开销的降低意味着性能的提高，所以预处理语句非常适合用来完成需要反复执行许多次的操作。比如说，在插入多个数据行时，只要先用 prepare()方法对 INSERT 语句进行一次预处理，就可以通过在一个循环里反复调用 execute()方法的办法去插入所有的数据行，只需使用占位符把值绑定到预处理语句即可。

9.1.7 利用占位符来处理带引号的数据值

在 PHP 里构造 SQL 语句字符串时，一定要足够重视引号问题，就像在使用 C 和 Perl 等其他语言来开发 MySQL 脚本时一样。假设你正在构造一条用来把一条新行插入到数据表里的语句。在语句字符串里，你可能给那些将被插入到字符串数据列里的值都加上了引号，如下所示：

```
$last = "O'Malley";
$first = "Brian";
$expiration = "2013-09-01";
$stmt = "INSERT INTO member (last_name,first_name,expiration)"
      . " VALUES('$last','$first','$expiration')";
```

这里的问题是有一项带双引号的数据（O'Malley）本身就带有一个单引号，如果把这条语句发送给 MySQL 服务器，就会导致一个语法错误。为了解决这一问题，在 C 程序里可以调用 mysql_real_escape_string()或 mysql_escape_string()函数；在 Perl DBI 脚本里可以调用 quote()函数。在 PHP 脚本里，可以调用 PDO 为数据库句柄提供的 quote()方法。比如说，调用函数 quote("O'Malley")将返回字符串值'O\'Malley'。如果在构造语句时使用了 quote()方法，直接把该方法的返回值插入到语句字符串里即可，用不着添加额外的引号：

```
$last = $dbh->quote ("O'Malley");
$first = $dbh->quote ("Brian");
$expiration = $dbh->quote ("2013-09-01");
$stmt = "INSERT INTO member (last_name,first_name,expiration)"
      . " VALUES($last,$first,$expiration)";
```

不过，与 DBI 模块里的同名函数相比，PDO 的 quote()方法还存在一些不足，限制了它的使用。

- 有几种驱动程序还没有实现这个方法，若它们实现了，它将返回 FALSE 而不是一个括在引号

里的字符串。

- 对于 NULL 值，它在 SQL 语句字符串里应该是一个不带任何引号的单词“NULL”，但如果把 NULL 传递给 quote()方法，它将返回空字符串（' '）。为了解决这个问题，必须知道值到底是什么，或者对它进行测试并根据它是否表示 NULL 值来进行不同的处理。这听起来容易，做起来难。

因为这些不足，我个人认为应该尽量避免使用 quote()方法，除非你已经确切地知道你将要处理的字符串数据都是非 NULL 值。一个更好的办法是使用预处理语句。然后在 SQL 语句里安排一些占位符，让 PDO 替你处理好所有引号问题。在预处理阶段，用“？”字符在 SQL 语句里占个位子——把它放到你想让数据值出现的位置上。在执行阶段，以参数数组的形式把数据值提供给该语句：

```
$stmt = "INSERT INTO member (last_name,first_name,expiration) VALUES(?,?,?)";
$sth = $dbh->prepare ($stmt);
$sth->execute (array ("O'Malley", "Brian", "2013-09-01"));
```

PDO 不仅能对字符串里的特殊字符进行必要的处理，还能对数值和 NULL 等非字符串值进行正确的处理。

提供数据值的另一个办法是在调用 execute()方法之前用 bindValue()方法把它们分别绑定到相应的占位符上：

```
$stmt = "INSERT INTO member (last_name,first_name,expiration) VALUES(?,?,?)";
$sth = $dbh->prepare ($stmt);
$sth->bindValue (1, "O'Malley");
$sth->bindValue (2, "Brian");
$sth->bindValue (3, "2013-09-01");
$sth->execute ();
```

前面几个例子使用的都是位置型占位符，它们都是同样的“?”字符，区别只在于它们在语句字符串里的位置。PDO 还支持名字型占位符：占位符由一个紧跟在一个冒号字符后面的名字构成。对将要执行的语句进行预处理，然后向 execute()方法传递一个关联数组，该数组把每一个值和相应的名字关联在了一起：

```
$stmt = "INSERT INTO member (last_name,first_name,expiration)
         VALUES(:last_name,:first_name,:expiration)";
$sth = $dbh->prepare ($stmt);
$sth->execute (array (
                 ":last_name" => "O'Malley",
                 ":first_name" => "Brian",
                 ":expiration" =>"2013-09-01"
               ));
```

在调用 execute()方法之前，还可以先把每个值绑定到它的占位符名字上：

```
$stmt = "INSERT INTO member (last_name,first_name,expiration)
         VALUES(:last_name,:first_name,:expiration)";
$sth = $dbh->prepare ($stmt);
$sth->bindValue (":last_name", "O'Malley");
$sth->bindValue (":first_name", "Brian");
$sth->bindValue (":expiration", "2013-09-01");
$sth->execute ();
```

名字型占位符的一个优点是占位符和数据值之间的关联很清晰，尤其适用于有大量参数的场合。

9.1.8 出错处理

在和 MySQL 交互时，出错处理的安排非常关键。如果想当然地认为每一个调用都会成功，在真的发生错误时将会很难找到脚本不工作的原因。

如果调用 new PDO()去连接数据库服务器，它在失败时将抛出一个异常，在成功时将返回一个合法的数据库句柄。如果基于这个句柄的后续 PDO 操作发生了错误，PDO 将根据当前的 PDO 出错处理模式对它们进行处理。你可以用下面这条语句来设置出错处理模式：

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, mode_value);
```

PDO 支持 3 种出错处理模式值。

- ❑ PDO::ERRMODE_SILENT。PDO 将只为引起错误的对象设置出错信息，其他什么事情都不做。这是默认的出错处理模式。
- ❑ PDO::ERRMODE_WARNING。这类似于 PDO::ERRMODE_SILENT 模式，但 PDO 在设置出错信息以外还将抛出一条警告消息。
- ❑ PDO::ERRMODE_EXCEPTION。在设置出错信息之后，PDO 还将抛出一个异常。

在以上 3 种出错处理模式下，如果你确切地知道是哪个对象发生了错误，就可以调用该对象的 errorCode()或 errorInfo()方法来获取出错信息。

- ❑ errorCode()返回一个由 5 个字符构成的 SQLSTATE 值。如果返回值是 PDO::ERR_NONE ('00000')，则表明没有发生任何错误。
- ❑ errorInfo()返回一个由 3 个元素构成的数组，包括一个 SQLSTATE 值、一个与驱动程序有关的出错代码和一条出错消息。对于 MySQL，后两个值是一个数值形式的出错代码和一条描述性的出错消息。

在默认模式和警告模式下，出错处理意味着需要对每一个有可能失败的 PDO 操作的结果进行检查。比如说：

```
if (!($sth = $dbh->prepare ("SELECT * FROM non_existent_table")))
  die ("Cannot prepare statement: " . $dbh->errorCode () . "\n");
else if (!$sth->execute ())
  die ("Cannot execute statement: " . $sth->errorCode () . "\n");
```

请注意，errorCode()方法是用那个发生了错误的句柄来调用的。errorInfo()方法也要这样来调用。

如果启用了异常模式，PHP 将在某个 PDO 操作发生错误时抛出一个 PDOException。如果没有发生任何错误，操作将成功；否则该错误抛出的异常将导致 PHP 结束整个脚本的执行，除非你已经安排了一些代码来捕获它。如果你想捕获因错误而抛出的异常，必须把可能失败的代码放在一个 try 语句块里，把相应的出错处理代码放在对应的 catch 语句块里，如下所示：

```
try
{
  # ...perform a database operation...
}
catch (PDOException $e)
{
  # ...handle the error...
}
```

异常对象（本例中的$e）有它自己的方法来提供出错信息。

- getCode()方法：返回一个出错代码。
- getMessage()方法：返回一个包含出错消息的字符串。

下面的例子启用了 PDO::ERRMODE_EXCEPTION 模式，并演示了如何在语句执行失败时显示出错信息：

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
try
{
  $dbh->exec ("DELETE FROM non_existent_table");
}
catch (PDOException $e)
{
  # Print error information from exception object
  print ("getCode value: " . $e->getCode () . "\n");
  print ("getMessage value: " . $e->getMessage () . "\n");
  # Print error information from database handle
  print ("errorCode value: " . $dbh->errorCode () . "\n");
  print ("errorInfo value: " . join (",", $dbh->errorInfo ()) . "\n");
}
```

这个例子将显示来自异常对象（$e）和来自数据库句柄对象（$dbh）的出错信息，这是因为我们在 try 语句块里只用到了$dbh 这一个 PDO 句柄。如果在 try 语句块里使用了多个 PDO 句柄，就无法得知 catch 语句块里到底是哪个句柄引起的错误，只能依赖异常对象的方法。不过，你可以重新构造代码，给每个 PDO 句柄分别安排一组它自己的 try/catch 语句块。

有些 PHP 函数或操作在发生错误时不仅会返回一个状态值，还会生成一条出错消息。在 Web 上下文里，这种出错消息将出现在被发送到客户浏览器的页面里。如果不想让用户看到这种出错消息，可以在函数名的前面加上一个“@”操作符。比如说，如果你想阻制一个名为 some_func()的函数在发生错误时向用户显示出错消息（因为你打算用一种更适当的方式来报告错误），只要按下面这样去做就行了：

```
$status = @some_func ();
```

9.2　PHP 脚本实战

本章的后续内容将解决我们在第 1 章里提出的一些目前仍未完成的目标。

- 考试记分项目：编写一个用来录入和修改考试与测验分数的脚本。
- 在美国历史研究会的 Web 站点上向访问者提供一个关于美国总统生平事迹的小测验。将小测验做成交互式的，测验题要即时生成。
- 使美国历史研究会的会员能够以在线方式修改他们的会员名录资料。这一方面能使信息保持最新，同时也减少了研究会秘书在这方面的工作量。

每个脚本都会生成多个彼此关联的 Web 页面，所以这些脚本在它们各自的前后两次调用中需要通过嵌在 Web 页面（由脚本创建）里的信息进行通信。如果你不熟悉 Web 页面之间的通信机制，请参阅 8.2.4 节下的第 4 小节。

9.2.1　考试分数的在线录入

在这一节，我们将把注意力集中到考试记分项目，编写一个 score_entry.php 脚本来管理学生

们的分数。我们将把与该项目有关的 Web 页面全都放在 Apache 文档树的 gp 子目录里，这个子目录在我们的站点上对应于下面这个 URL：

```
http://localhost/gp/
```

这个子目录现在还空无一物，请求这个 URL 的访问者目前还只能看到一条“Page not found”（网页未找到）信息或者一个空白的目录清单页面，所以当务之急是要在 gp 子目录里创建一个简短的 `index.php` 脚本来充当考试记分项目的主页。下面这个脚本足以应付眼前的需要。这个脚本生成的 Web 页面里有两个链接，一个指向我们在 8.4.5 节为考试记分项目编写的 `score_browse.pl` 脚本，另一个指向我们马上就要编写的 `score_entry.php` 脚本：

```
<?php
# Grade-Keeping Project home page

require_once "sampdb_pdo.php";

$title = "Grade-Keeping Project";
html_begin ($title, $title);
?>

<p>
<a href="/cgi-bin/score_browse.pl">View</a> test and quiz scores
</p>
<p>
<a href="score_entry.php">Enter or edit</a> test and quiz scores
</p>

<?php
html_end ();
?>
```

可以在 sampdb 发行版本的 phpapi/gp 子目录里找到这个 `index.php` 脚本，把它复制到 Web 服务器文档树中的 gp 子目录里去。

9

我们将要设计和实现的 `score_entry.php` 脚本将具备两大功能，一是录入一组考分，二是使我们能够修改现有的各组考分。它提供的录入功能将把分数添加到数据库里，而编辑功使我们能在今后修改分数，比如把因生病或其他原因而缺考的学生的补考成绩录入数据库，或者改正我们输错的考试成绩，等等。下面是考分录入脚本的概念性框架。

- ❑ 在初始页面上将显示一份已知考试事件的清单，你既可以点选一个现有事件，也可以提出要创建一个新的考试事件。
- ❑ 如果你选择的是创建一个新的事件，脚本将在下一个页面里要求你给出一个日期和类型（考试还是测验）。在把新事件添加到数据库之后，脚本将重新显示事件清单页面，你新添加的事件也将出现在其中。
- ❑ 如果你从清单里选择了一个现有事件，脚本将显示一个考分录入页面，显示事件的 ID、日期、种类、一份学生名单和一个 Submit（提交）按钮。学生名单的每一行列出了每位学生的姓名和他这次的分数。如果你选择的是一个新事件，考分将都是空白。如果你选择的是一个现有的事件，考分就将是你在此前录入的那些数字。当你录入或者修改完考分后，单击 Submit 按钮，脚本就会把考分输入到 `score` 数据表或者修改现有分数。这个操作需要作为事务来完成，确保在发生错误时，所有作为部分事务或全部事务完成的考分修改都被取消。

1. 收集PHP中的Web输入

在实现 score_entry.php 脚本之前，先讨论一下输入参数在 PHP 脚本里的工作原理。这个脚本需要完成好几个不同的操作，这意味着它必须在页面之间传递一个状态值，告诉脚本在每次执行过程中应该做什么事情。传递状态值的办法之一是在 URL 地址的尾部追加一个参数。比如说，我们可以像下面这样在这个脚本的 URL 地址尾部加上一个名为 action 的参数：

```
http://localhost/gp/score_entry.php?action=value
```

参数值也可以来自用户提交的某个表单的内容。作为表单提交的组成部分，用户的浏览器返回的表单里的每个字段都有一个名字和一个值。

PHP 把输入参数放在几个特殊的数组里供脚本使用。被编码在 URL 地址尾部以及通过 GET 方法发送回来的参数将被放在$HTTP_GET_VARS 全局数组和$.PUT 超全局数组里。

全局数组必须经过明确的声明才能在 PHP 脚本的非顶层上下文（例如函数定义的内部）里使用。超全局数组在任何一个层次里都是无需经过任何特殊的声明就可以访问的。我们将使用$_GET 和$_PUT 这两个超全局数组。（$HTTP_GET_VARS 和$HTTP_POST_VARS 现在已经有点儿过时了。）

$_GET 和$_PUT 是关联数组，数组元素的键就是各参数的名字。比如说，如果 URL 地址字符串里嵌有一个 action 参数，我们在 PHP 脚本里就可以通过$_GET ["action"]变量去访问它的值。假设某个表单包含着 name 和 address 两个字段，当用户提交这个表单时，Web 服务器将调用一个脚本去处理这个表单的内容。如果这个表单是作为一个 get 请求被提交的，我们在脚本里只需查看$_GET ["name"]和$_GET ["address"]变量就可以知道用户在表单里输入的是什么值。如果这个表单是作为一个 post 请求被提交的，就应该去查看$_POST ["name"]和$_POST ["address"]变量。

如果表单包含的字段很多，给它们分别起一个独一无二的名字恐怕没那么方便。PHP 也支持参数数组的传递，而且用起来很容易。如果把字段命名为 x[0]、x[1]等，PHP 将把它们保存在$_GET["x"]或$_POST["x"]变量里，而这个变量的值是另一个数组。如果把这个数组值赋值给变量$x，就可以通过$x[0]、$x[1]等去访问参数数组的元素。

在大多数场合，我们用不着关心某个参数是作为 get 请求还是作为 post 请求被提交的，因为我们可以编写一个这样的 script_param()实用工具例程，给它一个参数名，让它在那两个数组里都去找一下这个参数的值。如果在两处都没有找到，则返回 NULL：

```
function script_param ($name)
{
  $val = NULL;
  if (isset ($_GET[$name]))
    $val = $_GET[$name];
  else if (isset ($_POST[$name]))
    $val = $_POST[$name];
  if (get_magic_quotes_gpc ())
    $val = remove_backslashes ($val);
  return ($val);
}
```

不管输入参数会存放在哪个数组里，script_param()函数可以让脚本简单地通过参数名直接访问它们。在提取出一个参数值以后，它还要把这个参数值传递到 remove_backslashes()函数里去。这是为了适应 PHP 初始化文件用下面这条语句把配置选项 magic_quotes_gpc 设置为激活状态：

```
magic_quotes_gpc = On;
```

如果这个选项被激活，PHP 就会添加反斜线字符以引号或反斜线等特殊字符。这些额外的反斜线字符会增加我们检查参数值是否合法的难度，所以要用 remove_backslashes()函数把它们去除掉。在 PHP 里，参数有可能被返回为多层嵌套的数组，所以这里使用了递归算法，如下所示。

```
function remove_backslashes ($val)
{
  if (is_array ($val))
  {
    foreach ($val as $k => $v)
      $val[$k] = remove_backslashes ($v);
  }
  else if (!is_null ($val))
    $val = stripslashes ($val);
  return ($val);
}
```

Web 输入参数与 register_globals

读者对 PHP 的配置选项 register_globals 可能不会陌生，它的作用是使来自 Web 的输入参数直接被注册为脚本里的变量。比如说，一个名为 x 的表单字段或 URL 参数将直接被保存到你脚本中一个名为$x 的变量里。这一做法虽然方便，但同时也意味着客户可以直接在你的脚本里创建出一些不符合你意愿的变量来。这是一种安防风险，所以 PHP 开发者们建议最好禁用 register_globals 选项。在编写 script_param()例程时，我特意只使用了专为输入参数而准备的数组，这既增加了安全性，也使它在 register_globals 选项的任何一种设置情况下都能够工作。

2. 显示和输入分数

既然我们已经能提取 Web 输入参数，那么我们就可以编写 score_entry.php 脚本了。这个脚本需要在它自己的前后两次调用中交流一些信息。我们将使用一个名为 action 的参数来做这件事情，执行脚本时，这个参数的值可以这样获得：

```
$action = script_param ("action");
```

如果没有给出这个参数，就说明这是脚本的首次调用；否则，脚本将根据变量$action 的值来决定将要做哪些事情。下面是 score_entry.php 脚本的基本框架：

```
<?php
# score_entry.php - Score Entry script for grade-keeping project

require_once "sampdb_pdo.php";

# define action constants
define ("SHOW_INITIAL_PAGE", 0);
define ("SOLICIT_EVENT", 1);
define ("ADD_EVENT", 2);
define ("DISPLAY_SCORES", 3);
define ("ENTER_SCORES", 4);

# ...put input-handling functions here...

$title = "Grade-Keeping Project -- Score Entry";
html_begin ($title, $title);
```

```
$dbh = sampdb_connect ();

# determine what action to perform (the default is to
# present the initial page if no action is specified)

$action = script_param ("action");
if (is_null ($action))
  $action = SHOW_INITIAL_PAGE;

switch ($action)
{
case SHOW_INITIAL_PAGE:   # present initial page
  display_events ($dbh);
  break;
case SOLICIT_EVENT:       # ask for new event information
  solicit_event_info ();
  break;
case ADD_EVENT:           # add new event to database
  add_new_event ($dbh);
  display_events ($dbh);
  break;
case DISPLAY_SCORES:      # display scores for selected event
  display_scores ($dbh);
  break;
case ENTER_SCORES:        # enter new or edited scores
  enter_scores ($dbh);
  display_events ($dbh);
  break;
default:
  die ("Unknown action code ($action)\n");
}

$dbh = NULL;  # close connection

html_end ();
?>
```

变量$action 有好几种可能的值，我们用一个 switch 语句来测试它。PHP 语言里的 switch 语句与 C 语言里的 switch 很相似，它在这里的作用是判断脚本应该采取什么动作并调用函数完成这个动作。为了可以不必使用文本操作值，switch 语句里使用了一些符号动作名称，这些符号是在脚本的开头部分利用 PHP 语言的 define()构造定义的。

下面依次分析处理这些动作的函数。第一个，即 display_events()函数，将显示一个事件清单，我们是通过发出 MySQL 查询命令去检索 grade_event 数据表的数据行来获得和显示有关信息的。这份清单里的每一行列出了相应的事件 ID、日期和事件类型（考试或测验）。Web 页面里的事件 ID 都是一些超链接，选择它们就能对给定事件中的分数进行编辑了。此外，在事件行的后面，display_events()函数又添加了一个用来创建新事件的超链接。下面是这个函数的代码：

```
function display_events ($dbh)
{
  print ("Select an event by clicking on its number, or select\n");
```

```
  print ("New Event to create a new grade event:<br /><br />\n");
  print ("<table border=\"1\">\n");

  # Print a row of table column headers

  print ("<tr>\n");
  display_cell ("th", "Event ID");
  display_cell ("th", "Date");
  display_cell ("th", "Category");
  print ("</tr>\n");

  # Present list of existing events.  Associate each event id with a
  # link that will show the scores for the event.

  $stmt = "SELECT event_id, date, category
           FROM grade_event ORDER BY event_id";
  $sth = $dbh->query ($stmt);

  while ($row = $sth->fetch ())
  {
    print ("<tr>\n");
    $url = sprintf ("%s?action=%d&event_id=%d",
                    script_name (),
                    DISPLAY_SCORES,
                    $row["event_id"]);
    display_cell ("td",
                  "<a href=\"$url\">"
                    . $row["event_id"]
                    . "</a>",
                  FALSE);
    display_cell ("td", $row["date"]);
    display_cell ("td", $row["category"]);
    print ("</tr>\n");
  }

  # Add one more link for creating a new event

  print ("<tr align=\"center\">\n");
  $url = sprintf ("%s?action=%d",
                  script_name (),
                  SOLICIT_EVENT);
  display_cell ("td colspan=\"3\"",
                "<a href=\"$url\">Create New Event</a>",
                FALSE);
  print ("</tr>\n");

  print ("</table>\n");
}
```

超链接将导致 score_entry.php 脚本被再次调用。我们在构造这些超链接的 URL 地址时使用了一个名为 script_name()的函数，它的用途是确定脚本自身的路径名。有了 script_name()函数，我们就不必把脚本的文件名硬编码在代码里（如果你把脚本的文件名硬编码在代码里而你以后又重新命名了这个文件，这个脚本就无法正常工作了），在 sampdb_pdo.php 文件中可以找到 script_name。

类似于 script_param()函数，script_name()函数也需要访问几个 PHP 全局数组。但这两个函数使用的数组是不同的，因为脚本的文件名是 Web 服务器所提供的信息的一部分，在输入参数里是找不到的。下面是 script_name()函数的代码：

```
function script_name ()
{
  return ($_SERVER["SCRIPT_NAME"]);
}
```

display_events()函数又调用了一个子函数 display_cell()来生成 event 表里的各项数据：

```
# Display a cell of an HTML table.  $tag is the tag name ("th" or "td"
# for a header or data cell), $value is the value to display, and
# $encode should be true or false, indicating whether or not to perform
# HTML-encoding of the value before displaying it.  $encode is optional,
# and is true by default.

function display_cell ($tag, $value, $encode = TRUE)
{
  if (strlen ($value) == 0) # is the value empty/unset?
    $value = " ";
  else if ($encode) # perform HTML-encoding if requested
    $value = htmlspecialchars ($value);
  print ("<$tag>$value</$tag>\n");
}
```

如果你在 display_events()函数生成的表格里选了“Create New Event”（创建新事件）链接，score_entry.php 脚本就将被再次调用并去完成 SOLICIT_EVENT 动作。这个动作将触发 solicit_event_info()函数的调用，这个函数会显示一个表单供你输入新事件的日期和类型。下面是 solicit_event_info()函数的代码：

```
function solicit_event_info ()
{
  printf ("<form method=\"post\" action=\"%s?action=%d\">\n",
          script_name (),
          ADD_EVENT);
  print ("Enter information for new grade event:<br /><br />\n");
  print ("Date: ");
  print ("<input type=\"text\" name=\"date\" value=\"\" size=\"10\" />\n");
  print ("<br />\n");
  print ("Category: ");
  print ("<input type=\"radio\" name=\"category\" value=\"T\"");
  print (" checked=\"checked\" />Test\n");
  print ("<input type=\"radio\" name=\"category\" value=\"Q\" />Quiz\n");
  print ("<br /><br />\n");
  print ("<input type=\"submit\" name=\"submit\" value=\"Submit\" />\n");
  print ("</form>\n");
}
```

solicit_event_info()函数生成的表单里有一个供你输入日期的输入框、两个供你选择新事件是一次考试还是一次测验的按钮和一个 Submit 按钮。默认的事件类型是'T'，即考试（脚本编写了文本 HTML 来构建这里的表单。对于本章后面的脚本，我们将开发一些函数来生成表单元素）。

当你填写好这份表单并提交它时，score_entry.php 脚本将被再次调用并去完成 ADD_EVENT 动

作。add_new_event()函数将被调用以在 grade_event 表中输入新行：

```
function add_new_event ($dbh)
{
  $date = script_param ("date");          # get date and event category
  $category = script_param ("category");  # entered by user

  if (empty ($date))  # make sure a date was entered, and in ISO 8601 format
    die ("No date specified\n");
  if (!preg_match ('/^\d{4}\D\d{1,2}\D\d{1,2}$/', $date))
    die ("Please enter the date in ISO 8601 format (CCYY-MM-DD)\n");
  if ($category != "T" && $category != "Q")
    die ("Bad event category\n");

  $stmt = "INSERT INTO grade_event (date,category) VALUES(?,?)";
  $sth = $dbh->prepare ($stmt);
  $sth->execute (array ($date, $category));
}
```

add_new_event()先调用 script_param()库例程来访问新事件创建表单里的 date 和 type 字段里给出的参数值。然后，为安全起见，进行了以下几项小检查。

- 日期参数值不允许为空，并且必须按 ISO 8601 格式输入。preg_match()函数通过模式匹配判断日期值是否符合 ISO 8601 格式，如下所示：

  ```
  preg_match ('/^\d{4}\D\d{1,2}\D\d{1,2}$/', $date)
  ```

  这里使用了单引号以防止美元字符（$）和反斜线字符（\）被解释为特殊字符。如果日期值由 3 组以非数字字符分隔的数字序列构成，这个测试的结果就将为真。虽说它无法做到万无一失，但这个测试很容易添加到脚本里，并且它也确实能捕获不少的常见错误。

  为了更加安全，需要在插入数据前设置 SQL 模式以启用数据限制，例如：

  ```
  $dbh->exec ("SET sql_mode = 'TRADITIONAL'");
  ```

- 事件类型必须是 grade_event 数据表中的 category 数据列允许使用的值（即'T'或'Q'）。

如果参数值满足以上条件，add_new_event()函数就会把一个新行插入到 grade_event 数据表中。在构造查询命令字符串的时候，语句执行代码使用了占位符来确保插入到查询字符串里的各数据不会导致引号问题。在构造完语句后，add_new_event()函数先返回到了脚本的主体（switch 语句），它显示事件清单，因而你可以选择事件并开始输入分数。

当你在 display_events()函数生成的事件清单页面里选择某项时，score_entry.php 脚本将调用 display_scores()函数。事件链接里的事件 ID 将被编码为一个 event_id 参数，display_scores()函数将提取出这个参数值，在确认它是一个整数后，用它构造一个查询来检索每个学生的记录和参加此次事件的学生的分数。下面是 display_scores()函数的代码：

```
function display_scores ($dbh)
{
  # Get event ID number, which must look like an integer
  $event_id = script_param ("event_id");
  if (!ctype_digit ($event_id))
    die ("Bad event ID\n");

  # Select scores for the given event
  $stmt = "
```

```
    SELECT
      student.student_id, student.name, grade_event.date,
      score.score AS score, grade_event.category
    FROM student
      INNER JOIN grade_event
      LEFT JOIN score ON student.student_id = score.student_id
                      AND grade_event.event_id = score.event_id
    WHERE grade_event.event_id = ?
    ORDER BY student.name";
  $sth = $dbh->prepare ($stmt);
  $sth->execute (array ($event_id));

  # fetch the rows into an array so we know how many there are
  $rows = $sth->fetchAll ();
  if (count ($rows) == 0)
    die ("No information was found for the selected event\n");

  printf ("<form method=\"post\" action=\"%s?action=%d&event_id=%d\">\n",
          script_name (),
          ENTER_SCORES,
          $event_id);

  # print scores as an HTML table

  for ($row_num = 0; $row_num < count ($rows); $row_num++)
  {
    $row = $rows[$row_num];
    # Print event info and table heading preceding the first row
    if ($row_num == 0)
    {
      printf ("Event ID: %d, Event date: %s, Event category: %s\n",
              $event_id,
              $row["date"],
              $row["category"]);
      print ("<br /><br />\n");
      print ("<table border=\"1\">\n");
      print ("<tr>\n");
      display_cell ("th", "Name");
      display_cell ("th", "Score");
      print "</tr>\n";
    }
    print ("<tr>\n");
    display_cell ("td", $row["name"]);
    $col_val = sprintf ("<input type=\"text\" name=\"score[%d]\"",
                        $row["student_id"]);
    $col_val .= sprintf (" value=\"%d\" size=\"5\" /><br />\n",
                         $row["score"]);
    display_cell ("td", $col_val, FALSE);
    print ("</tr>\n");
  }

  print ("</table>\n");
  print ("<br />\n");
  print ("<input type=\"submit\" name=\"submit\" value=\"Submit\" />\n");
  print "</form>\n";
}
```

display_scores()函数里构造了一个对选中事件进行分数检索的查询，这个查询并不是一个简单的数据表联结，因为那样会漏掉那些没有参加这次事件的学生。再进一步讲，如果你选取的是一个新事件，这种联结将选不到任何记录，你看到的将是一个空成绩表。因此，要想一个不少地把每位学生的记录都检索出来，而不管 score 数据表里是否有他的考分，我们必须使用 LEFT JOIN 操作。如果某个学生没有某次事件的考分，他在查询结果里就将是 NULL。（它与 2.7.3 节中 display_scores()函数用来检索分数行的查询。）

脚本将检索到分数作为输入域放入表单中，这些输入域的名字是 score[n]，n 是学生的学号（即一个 student_id 值）。当你在完成了考分的录入或修改工作之后，提交表单将使它们保存在数据库中。当浏览器把这个表单发送回 Web 服务器时，PHP 将把这些输入域转换为与 score 名称相关联的元素，如下所示：

```
$score = script_param ("score");
```

这个数组的元素以学生 ID 号作为键，所以很容易把每个学生与通过表单提交来的相应的考试分数关联起来。对这个表单进行处理可能需要执行多条语句（每个学生一条），而我们不希望只有一部分语句执行成功。在第 1 章中，我们把 score 数据表创建为一个 InnoDB 数据表。因此而能够受益于 InnoDB 的事务处理能力。就眼前的问题而言，通过把整个数据录入操作作为一个事务去完成，我们可以确保它将作为一个不可分割的原子化动作而发生。要么数据修改全部成功，要么数据库不发生任何变化。在 PDO 脚本里进行事务处理有一个如下所示的通用结构（其前提是已为 PDO 错误启用异常捕获机制）：

```
try
{
  $dbh->beginTransaction ();          # start the transaction
  # ... perform database operation ...
  $dbh->commit ();                    # transaction succeeded
}
catch (PDOException $e)
{
  $dbh->rollback ();                  # transaction failed
}
```

score_entry.php 脚本使用了上述结构来保证数据录入操作的整体性。（把回滚操作放在它自己的 try/catch 块里是为了防止它执行失败时结束脚本的运行。）

enter_scores()函数负责处理表单的内容，确定哪些考试分数需要更新或删除：

```
function enter_scores ($dbh)
{
  # Get event ID number and array of scores for the event

  $event_id = script_param ("event_id");
  $score = script_param ("score");

  if (!ctype_digit ($event_id)) # must look like integer
    die ("Bad event ID\n");

  # Prepare the statements that are executed repeatedly
  $sth_del = $dbh->prepare ("DELETE FROM score
                             WHERE event_id = ? AND student_id = ?");
```

9

```
  $sth_repl = $dbh->prepare ("REPLACE INTO score
                              (event_id,student_id,score)
                              VALUES(?,?,?)");

  # enter scores within a transaction
  try
  {
    $dbh->beginTransaction ();

    $blank_count = 0;
    $nonblank_count = 0;
    foreach ($score as $student_id => $new_score)
    {
      $new_score = trim ($new_score);
      if (empty ($new_score))
      {
        # if no score is provided for student in the form, delete any
        # score the student may have had in the database previously
        ++$blank_count;
        $sth = $sth_del;
        $params = array ($event_id, $student_id);
      }
      else if (ctype_digit ($new_score)) # must look like integer
      {
        # if a score is provided, replace any score that
        # might already be present in the database
        ++$nonblank_count;
        $sth = $sth_repl;
        $params = array ($event_id, $student_id, $new_score);
      }
      else
      {
        throw new PDOException ("invalid score: $new_score");
      }
      $sth->execute ($params);
    }
    # transaction succeeded, commit it
    $dbh->commit ();
    printf ("Number of scores entered: %d<br />\n", $nonblank_count);
    printf ("Number of scores missing: %d<br />\n", $blank_count);
  }
  catch (PDOException $e)
  {
    printf ("Score entry failed: %s<br />\n",
            htmlspecialchars ($e->getMessage ()));
   # roll back, but use empty exception handler to catch rollback failure
   try
   {
     $dbh->rollback ();
   }
   catch (PDOException $e) { }
  }
  print ("<br />\n");
}
```

学生 ID 值和与它们相关联的考试分数是通过遍历$score 数组而获得的。下面是在循环里处理每个考试分数的流程。

- 如果某个考试分数在去掉它的尾缀空格后是空格，就表明用户没有输入任何东西。但说不定以前曾经有过一个考试分数，所以脚本将尝试删除它。（比如说，曾错误地为某个学生输入了一个考试分数，可他实际上并没有参加考试，所以现在需要删除它。）如果该学生没有任何考试分数，DELETE 语句将不会找到任何应该删除的数据行，但那不会导致任何不良后果。
- 如果考试分数不是空格，这个函数将进行一些基本的输入检查，如果它看上去像是一个整数，则接受它。请注意，整数测试是用一个模式匹配操作而不是使用 PHP 的 is_int()函数完成的。后者用来测试某个变量的类型是不是整数，但表单值都是编码为字符串的。is_int()函数在遇到任何字符串时都将返回 FALSE，哪怕它只包含数字字符也会如此。这里需要的是对字符串进行内容检查。如果字符串$str 从头到尾的每一个字符都是数字，下面这个函数将返回 TRUE：

  ```
  ctype_digit ($str)
  ```

  如果考试分数没什么问题，将把它添加到 score 数据表里。这里之所以使用 REPLACE 语句而不是 INSERT 语句，是因为我们可能是在替换一个已经存在的考试分数而不是输入一个新的。如果某个学生在这次考试事件里没有任何分数，REPLACE 语句将添加一个新数据行，就像 INSERT 语句一样。否则，REPLACE 语句将用这次新输入的考试分数替换掉现有的。

在进入这个循环之前，脚本调用 beginTransaction()函数关闭了自动提交模式。在离开这个循环后，如果没有发生错误，脚本将提交本次事务。如果发生了错误，脚本将回滚本次事务。

score_entry.php 脚本到这里也就介绍得差不多了。现在，所有考试分数的录入和编辑工作都可以在 Web 浏览器里完成。一个明显的不足是这个脚本没有提供任何安全措施，能连接上 Web 服务器的任何人都可以修改考试分数。在稍后为历史研究会会员资料编写的脚本里，我们将演示一种简单的验证机制，该机制也可以用在这个脚本里。

9.2.2 创建一个交互式在线测验

历史研究会网站的目标之一是提供一个在线版的“美国历史事件”知识测验，其形式类似于该学会在其读者交流栏目的儿童专区里发布的小问答。我们精心创建的 president 数据表很适合用来作为这种历史知识小测验的试题来源。现在就来做这件事，这次使用的是一个名为 pres_quiz.php 的脚本。

这里的基本思路是随机挑选并提出一个关于某位总统的问题，然后请用户输入一个答案并检查那个答案是否正确。可以让脚本根据 president 数据表里的信息提出各种各样的问题。但为简明起见，把它限制为只提问某位总统出生在什么地方。另一项简化措施是出选择题。这对用户来说很方便，用户只要点选他认为正确的答案就行，用不着敲键盘。这给我们也减少了许多麻烦，因为不必构造任何匹式模板去分析用户到底输入了些什么，只要把用户选中的答案和我们从数据库查找出来的值比较一下就行了。

pres_quiz.php 脚本必须做好两件事。

- 在首次执行时，应该利用从 president 数据表里查到的信息生成并显示一道新的测验题。
- 在用户提交了一个答案之后，必须核对答案并向用户反馈它是否正确。如果答案不正确，脚本应该重新显示同样的测验题。否则，它应该生成并显示一道新的测验题。

这个脚本的整体思路相当简单。如果用户还没有提交过答案，则显示初始测验页面；如果用户已经提交了答案，则检查答案是否正确：

```
<?php
# pres_quiz.php - script to quiz user on presidential birthplaces

require_once "sampdb_pdo.php";

# ... put quiz-handling functions here ...

$title = "U.S. President Quiz";
html_begin ($title, $title);

$dbh = sampdb_connect ();

$response = script_param ("response");
if (is_null ($response))   # invoked for first time
  present_question ($dbh);
else                       # user submitted response to form
  check_response ($dbh);

$dbh = NULL;  # close connection

html_end ();
?>
```

为了构造一道测验题，我们将使用 ORDER BY RAND()语法。RAND()函数的作用是让我们能够从 president 数据表随机选取一个数据行。比如说，如下所示的查询命令将随机选取一位总统的名字和出生地：

```
SELECT CONCAT(first_name, ' ', last_name) AS name,
CONCAT(city, ', ', state) AS place
FROM president ORDER BY RAND() LIMIT 1;
```

总统的名字将被用来生成一道新的测验题："这位总统是在哪儿出生的？"他的出生地就是这道测验题的正确答案。我们还需要准备几个鱼目混珠的候选答案，它们可以用一个类似的查询命令来选取：

```
SELECT DISTINCT CONCAT(city, ', ', state) AS place
FROM president ORDER BY RAND();
```

我们将从这次查询的结果当中选出前 4 个与正确答案不一样的值。在这条查询命令使用 DISTINCT 是为了避免多次选中同一个出生地进入候选答案表。如果总统们的出生地各不相同的话，就没有必要使用这个 DISTINCT，只可惜它们有相同的。可以用下面这条语句核对一下：

```
mysql> SELECT city, state, COUNT(*) AS count FROM president
    -> GROUP BY city, state HAVING count > 1;
+-----------+-------+-------+
| city      | state | count |
+-----------+-------+-------+
| Braintree | MA    |     2 |
+-----------+-------+-------+
```

负责生成测验题和候选答案表的函数如下所示：

```
function present_question ($dbh)
{
  # issue statement to pick a president and get birthplace
  $stmt = "SELECT CONCAT(first_name, ' ', last_name) AS name,
           CONCAT(city, ', ', state) AS place
           FROM president ORDER BY RAND() LIMIT 1";
  $sth = $dbh->query ($stmt);
  $row = $sth->fetch ();
  $name = $row["name"];
  $place = $row["place"];

  # Construct the set of birthplace choices to present.
  # Set up the $choices array containing five birthplaces, one
  # of which is the correct response.
  $stmt = "SELECT DISTINCT CONCAT(city, ', ', state) AS place
           FROM president ORDER BY RAND()";
  $sth = $dbh->query ($stmt);
  $choices[] = $place;  # initialize array with correct choice
  while (count ($choices) < 5 && $row = $sth->fetch ())
  {
    if ($row["place"] != $place)
      $choices[] = $row["place"]; # add another incorrect choice
  }
  # randomize choices, display form
  shuffle ($choices);
  display_form ($name, $place, $choices);
}
```

present_question()函数将调用 display_form()函数生成测验题，它将把总统的名字、一组列出候选答案的单选按钮和一个 Submit 按钮显示在一个表单里。这个表单的基本用途是把测验题显示给用户，但它还需要做些别的事情。它必须把测验题显示给用户，还必须安排当用户提交答案后如何让发送回 Web 服务器的信息触发脚本，去检查那个答案是否正确，以及如何在那个答案不正确时重新显示测验题。

把测验题显示给用户的事情很好办，把总统的名字和候选的出生地显示在表单上就行了。检查答案的对错和重新显示测验题就有点儿棘手了。必须想个办法让脚本能够“记住”正确答案和重新生成测验题所需要的所有信息。办法之一是使用一组隐藏字段把所有必要的信息包含在表单里。这些字段同样是表单的组成部分，在用户提交答案时同样会被返回，但不会显示在用户眼前。

我们将使用 3 个名为 name、place 和 choices 的隐藏字段来分别保存总统姓名、正确的出生地和候选答案组。每组候选答案用 implode()函数编码为一个字符串，并在拼接字符串值时在它们之间插入一个特殊的分隔字符。（加上分隔符的目的是为了方便稍后在需要重新显示测验题时使用 explode()函数拆分字符串。）display_form()函数负责生成表单：

```
function display_form ($name, $place, $choices)
{
  printf ("<form method=\"post\" action=\"%s\">\n", script_name ());
  hidden_field ("name", $name);
  hidden_field ("place", $place);
  hidden_field ("choices", implode ("#", $choices));
  printf ("Where was %s born?<br /><br />\n", htmlspecialchars ($name));
```

```
  for ($i = 0; $i < 5; $i++)
  {
    radio_button ("response", $choices[$i], $choices[$i], FALSE);
    print ("<br />\n");
  }
  print ("<br />\n");
  submit_button ("submit", "Submit");
  print ("</form>\n");
}
```

display_form()函数使用了几个帮助函数来生成表单字段。第一个是 hidden_field()函数，它负责生成用于隐藏字段的<input>标记：

```
function hidden_field ($name, $value)
{
  printf ("<input type=\"%s\" name=\"%s\" value=\"%s\" />\n",
          "hidden",
          htmlspecialchars ($name),
          htmlspecialchars ($value));
}
```

因为 hidden_field()函数是一个在很多脚本里都有用武之地的通用例程，所以把它放到库文件 sampdb_pdo.php 里去。请注意，它还使用了 htmlspecialchars()函数来编码<input>标签的 name 和 value 属性，这么做是因为$name 和$value 变量的值有可能包含诸如引号之类的特殊字符。

下面是另外两个帮助函数 radio_button()和 submit_button()的代码：

```
function radio_button ($name, $value, $label, $checked)
{
  printf ("<input type=\"%s\" name=\"%s\" value=\"%s\"%s />%s\n",
          "radio",
          htmlspecialchars ($name),
          htmlspecialchars ($value),
          ($checked ? " checked=\"checked\"" : ""),
          htmlspecialchars ($label));
}

function submit_button ($name, $value)
{
  printf ("<input type=\"%s\" name=\"%s\" value=\"%s\" />\n",
          "submit",
          htmlspecialchars ($name),
          htmlspecialchars ($value));
}
```

当用户在候选答案当中挑选了一个出生地点并提交表单的时候，他选中的答案将作为 response 参数的值返回到 Web 服务器，只要调用 script_param()函数就可以知道 response 参数的值到底是什么。response 参数还将用来判断这是脚本的首次执行，还是用户正在提交此前显示在表单里的测验题的答案。因为这个参数在脚本首次执行时尚不存在，所以脚本的主体可以根据这个参数是否存在来确定应该做什么：

```
$response = script_param ("response");
if (is_null ($response))   # invoked for first time
  present_question ($dbh);
```

```
else                          # user submitted response to form
  check_response ($dbh);
```

还需要编写一个 check_response()函数去核对用户挑选的答案是否正确。这就需要用到 name、place 和 choices 隐藏字段里的值。我们已经把正确答案编码在了表单的 palce 字段里，而用户挑选的答案在 response 字段里，所以只要比较一下这两个字段的值就可以判断对错了。根据比较结果，check_response()函数将先提供一些反馈，然后生成并显示一道新的测验题，或者再次显示刚才那道测验题：

```
function check_response ($dbh)
{
  $name = script_param ("name");
  $place = script_param ("place");
  $choices = script_param ("choices");
  $response = script_param ("response");

  # Is the user's response the correct birthplace?

  if ($response == $place)
  {
    print ("That is correct!<br />\n");
    printf ("%s was born in %s.<br />\n",
            htmlspecialchars ($name),
            htmlspecialchars ($place));
    print ("Try the next question:<br /><br />\n");
    present_question($dbh);
  }
  else
  {
    printf ("\"%s\" is not correct.  Please try again.<br /><br />\n",
            htmlspecialchars ($response));
    $choices = explode ("#", $choices);
    display_form ($name, $place, $choices);
  }
}
```

任务完成。在历史研究会的主页上增加一个指向 pres_quiz.php 脚本的链接，就可以让其访问者通过这个小测验去了解一下自己的知识水平了。（可以把 index5.php 脚本文件从 sampdb 发行版本的 phpapi/ushl 子目录复制到 Web 服务器的文档树的 ushl 子目录，并把它重新命名为 index.php 以取代现有的同名文件。）

隐藏字段并不安全

在 pres_quiz.php 脚本里，我们使用了几个隐藏字段来传递它在下次执行时需要用到、但在本次执行时不应该让用户看到的信息。这对一个像它这样的趣味性脚本来说没什么问题，但在传递那些绝对不允许用户直接查看的信息时千万不要使用隐藏字段，因为它们其实毫无秘密可言。想知道为什么吗？把pres_quiz.php脚本安装到Web服务器文档树的ushl子目录并从浏览器去请求它，然后利用浏览器的“View Source”（查看源代码）命令去查看测验题页面的 HTML 源代码。你将发现 place 隐藏字段的内容，也就是这道小测验题的正确答案，就在那里等着你们去看呢。这意味

着这个小测验很容易作弊。单就这项特定的应用而言，这不是什么大问题，但这个例子无可辩驳地证明了隐藏字段一点儿也不安全。如果信息必须对用户严格保密，就应该选用其他一些真正安全的办法，比如在会话过程中把信息保存在服务器端等。

9.2.3　美国历史研究会：会员个人资料的在线修改

我们将要编写的最后一个 PHP 脚本是 edit_member.php，它将使“美国历史研究会”的会员们能够在线修改他们的会员名录资料。有了这个脚本，会员随时都可以改正或者更新他本人的资料而不必再向研究会递交申请报告。这能使会员信息保持最新，同时也减少了研究会的秘书的工作量。

很明显，必须确保会员资料只能由他本人或者研究会的秘书来修改。这意味着必须增加一些安防措施。作为一种简单的身份验证机制的演示，我们将使用 MySQL 来存放各位会员的口令，会员只有在提供了正确的口令之后才能访问脚本给出的修改表单。这个脚本的工作流程如下所示。

- 在首次调用中，edit_member.php 脚本将显示一个登录表单，会员必须在表单里输入自己的会员 ID 和口令。
- 当会员提交登录表单时，脚本将根据口令数据表来检查会员输入的 ID 和口令是否正确。如果口令正确，脚本将从 member 数据表里把这位会员的个人资料检索出来并显示在一个表单里供会员编辑。
- 当会员提交编辑表单时，我们将用这个表单里的内容更新数据库里的会员记录。

在开始上述工作之前，首先要分配口令。一个比较简便的办法是随机生成这些口令。下面两条语句将创建出一个名为 member_pass 的数据表，并为每位会员分配一个口令（从一个随机数生成 MD5 校验和，使用结果的前 8 个字符）。这里的脚本将采用这种简单而又快捷的办法，但在实际应用中，你应该让会员自己挑选一个口令：

```
mysql> CREATE TABLE member_pass (
    -> member_id INT UNSIGNED NOT NULL PRIMARY KEY,
    -> password CHAR(8));
mysql> INSERT INTO member_pass (member_id, password)
    -> SELECT member_id, LEFT(MD5(RAND()), 8) AS password FROM member;
```

除了要为 member 数据表里的每位会员生成一个口令外，还需要在 member_pass 数据表里增加一个特殊的“0 号会员”项，其口令相当于一位管理员（超级）用户的口令。研究会的秘书可以利用这个口令访问任何会员的记录：

```
mysql> INSERT INTO member_pass (member_id, password) VALUES(0, 'bigshot');
```

说明　在创建 member_pass 数据表之前，你应该把 samp_brows.pl 脚本移出你 Web 服务器的脚本子目录。8.4.4 节编写的这个脚本能让任何人查看 sampdb 数据库里任何数据表的内容，包括 member_pass 数据表。因此，它能用来查看会员或管理员的口令。

建立好 member_pass 数据表之后，我们就可以开始编写 edit_member.php 脚本了。下面是这个脚本的框架：

```
<?php
# edit_member.php - Edit U.S. Historical League member entries via the Web
```

```
require_once "sampdb_pdo.php";

# define action constants
define ("SHOW_INITIAL_PAGE", 0);
define ("DISPLAY_ENTRY", 1);
define ("UPDATE_ENTRY", 2);

# ... put input-handling functions here ...

$title = "U.S. Historical League -- Member Editing Form";
html_begin ($title, $title);

$dbh = sampdb_connect ();

# determine what action to perform (the default if
# none is specified is to present the initial page)

$action = script_param ("action");
if (is_null ($action))
  $action = SHOW_INITIAL_PAGE;

switch ($action)
{
case SHOW_INITIAL_PAGE:    # present initial page
  display_login_page ();
  break;
case DISPLAY_ENTRY:        # display entry for editing
  display_entry ($dbh);
  break;
case UPDATE_ENTRY:         # store updated entry in database
  update_entry ($dbh);
  break;
default:
  die ("Unknown action code ($action)\n");
}

$dbh = NULL;  # close connection

html_end ();
?>
```

display_login_page()函数生成初始页面，它将生成一个请求会员输入其 ID 和口令的表单：

```
function display_login_page ()
{
  printf ("<form method=\"post\" action=\"%s?action=%d\">\n",
          script_name (),
          DISPLAY_ENTRY);
  print ("Enter your membership ID number and password,\n");
  print ("then select Submit.\n<br /><br />\n");
  print ("<table>\n");
  print ("<tr>");
  print ("<td>Member ID</td><td>");
  text_field ("member_id", "", 10);
```

```
  print ("</td></tr>");
  print ("<tr>");
  print ("<td>Password</td><td>");
  password_field ("password", "", 10);
  print ("</td></tr>");
  print ("</table>\n");
  submit_button ("button", "Submit");
  print "</form>\n";
}
```

表单里的标题文字和输入框都被安排在一个 HTML 表格里，这是为了让它们排列得整齐些。因为这里只涉及两个字段，所以效果似乎不太明显。其实，这是一种非常有用的技巧，尤其是在你创建的表单里有多个不同长度的标题时，因为这让表单变得整齐。让表单组件对齐能便于用户阅读和理解。

display_login_page()函数还用到了两个辅助函数，它们都可以在 sampdb.php 库文件里找到。text_field()负责生成一个文本输入框：

```
function text_field ($name, $value, $size)
{
  printf ("<input type=\"%s\" name=\"%s\" value=\"%s\" size=\"%s\" />\n",
          "text",
          htmlspecialchars ($name),
          htmlspecialchars ($value),
          htmlspecialchars ($size));
}
```

除 type 属性的值是 password 以外，password_field()函数的代码与上面完全一样。

当用户输入会员 ID 和口令后提交这个表单时，action 参数的值将是 DISPLAY_ENTRY，所以 edit_member.php 脚本中的 switch 语句将在脚本的下一次调用中执行 display_entry()函数。display_entry()函数负责检查用户输入的口令是否正确：

```
function display_entry ($dbh)
{
  # Get script parameters; trim whitespace from the ID, but not
  # from the password, because the password must match exactly.

  $member_id = trim (script_param ("member_id"));
  $password = script_param ("password");

  if (empty ($member_id))
    die ("No member ID was specified\n");
  if (!ctype_digit ($member_id))                # must look like integer
    die ("Invalid member ID was specified (must be an integer)\n");
  if (empty ($password))
    die ("No password was specified\n");
  if (check_pass ($dbh, $member_id, $password)) # regular member
    $admin = FALSE;
  else if (check_pass ($dbh, 0, $password))     # administrator
    $admin = TRUE;
  else
    die ("Invalid password\n");

  $stmt = "SELECT
```

```
            last_name, first_name, suffix, email, street, city,
            state, zip, phone, interests, member_id, expiration
          FROM member WHERE member_id = ?
          ORDER BY last_name";
  $sth = $dbh->prepare ($stmt);
  $sth->execute (array ($member_id));

  if (!($row = $sth->fetch ()))
    die ("No user with member_id = $member_id was found\n");

  printf ("<form method=\"post\" action=\"%s?action=%d\">\n",
          script_name (),
          UPDATE_ENTRY);

  # Add member ID and password as hidden values so that next invocation
  # of script can tell which record the form corresponds to and so that
  # the user need not re-enter the password.

  hidden_field ("member_id", $member_id);
  hidden_field ("password", $password);

  # Format results of statement for editing

  print ("<table>\n");

  # Display member ID as static text

  display_column ("Member ID", $row, "member_id", FALSE);

  # $admin is true if the user provided the administrative password,
  # false otherwise. Administrative users can edit the expiration
  # date, regular users cannot.

  display_column ("Expiration", $row, "expiration", $admin);

  # Display other values as editable text

  display_column ("Last name", $row, "last_name");
  display_column ("First name", $row, "first_name");
  display_column ("Suffix", $row, "suffix");
  display_column ("Email", $row, "email");
  display_column ("Street", $row, "street");
  display_column ("City", $row, "city");
  display_column ("State", $row, "state");
  display_column ("Zip", $row, "zip");
  display_column ("Phone", $row, "phone");
  display_column ("Interests", $row, "interests");

  print ("</table>\n");

  submit_button ("button", "Submit");
  print "</form>\n";

}
```

display_entry()函数做的第一件事是验证口令。给定一个会员 ID 号，如果用户给出的口令与这位会员在 member_pass 数据表里的口令相匹配，或者与管理员口令（即那位特殊的“0 号会员”的口令）相匹配，edit_member.php 脚本就将把这位会员的个人资料显示在一个表单里供用户编辑。负责口令检查的 check_pass()函数将通过一个简单的查询从 member_pass 数据表里选取出一条记录，并把该记录中的 password 数据列值与用户在登录表单里输入的口令比较，如下所示：

```
function check_pass ($dbh, $id, $pass)
{
  $stmt = "SELECT password FROM member_pass WHERE member_id = ?";
  $sth = $dbh->prepare ($stmt);
  $sth->execute (array ($id));
  # TRUE if a password was found and it matches
  return (($row = $sth->fetch ()) && $row["password"] == $pass);
}
```

如果口令验证无误，display_entry()函数将从 member 数据表里检索出与给定会员 ID 对应的记录并用这条记录里的信息生成一个编辑表单。大多数字段都是供用户编辑的文本输入框，但这里有两个例外：首先，member_id 值将被显示为静态文本，因为它是唯一确定会员行的键值，不允许改变。其次，会员资格的失效日期也不允许会员修改（否则，如果会员本人把这个日期修改为一个距今更远的日期，他没有交纳会员年费就可延续自己的会员资格）。但是，如果用户在登录表单里给出了管理员口令，脚本就将把这个失效日期显示为可编辑字段。假设研究会的秘书知道这个口令，他们就可以替那些交纳了年费的会员延续会员资格失效日期。

display_column()函数负责显示字段标签和字段值。这个函数的输入参数依次为：$label，文本输入框旁边显示的标签；$row，用来存放待编辑记录的数组；$col_name，与该字段对应的数据列的名称；$editable，一个用来表明该字段显示为可编辑文本框还是静态文本的布尔值，它的默认值是 TRUE。下面就是 display_column()函数的代码：

```
function display_column ($label, $row, $col_name, $editable = TRUE)
{
  print ("<tr>\n");
  print ("<td>" . htmlspecialchars ($label) . "</td>\n");
  print ("<td>");
  if ($editable)  # display as editable field
    text_field ("row[$col_name]", $row[$col_name], 80);
  else            # display as read-only text
    print (htmlspecialchars ($row[$col_name]));
  print ("</td>\n");
  print ("</tr>\n");
}
```

对于可编辑值，display_column()函数将以 row[col_name]为名生成一个文本框。这样，用户提交这个表单时，PHP 就能把所有这些字段的值放到一个数组变量里，其中各个元素的下标就是它所对应的数据列名称。这能让我们很容易地把表单的内容提取出来，另外当我们更新数据库里的记录时，它能让我们方便地把字段值与 member 数据表里的各个数据列对应起来。比如说，假设你已经把数组提取到了变量$row 里，那么我们可以通过$row["phone"]访问电话号码。

display_entry()函数还通过两个隐藏字段把 member_id 和 password 值嵌在了表单里。这样，当用户提交编辑后的项时，我们就能把这两个参数值传递到 edit_member.php 脚本的下一次调用中。

ID 将告诉脚本需要更新 member 数据表里的哪一行，口令使脚本能够验证用户的身份以确保他就是刚才登录的那位会员。（注意，这个简单的身份验证机制将以明文的形式在服务器和浏览器之间传递口令，这通常不是一个好的解决方案。幸好“美国历史研究会”对安全性要求并不很高，这个方案已经足以满足应用要求。如果让你处理的是一些金融数据，就应该使用一种更安全的身份验证机制。）

以下函数负责在用户提交的表单后更新会员记录：

```
function update_entry ($dbh)
{
  # Get script parameters; trim whitespace from the ID, but not
  # from the password, because the password must match exactly,
  # or from the row, because it is an array.

  $member_id = trim (script_param ("member_id"));
  $password = script_param ("password");
  $row = script_param ("row");

  $member_id = trim ($member_id);
  if (empty ($member_id))
    die ("No member ID was specified\n");
  if (!ctype_digit ($member_id))            # must look like integer
    die ("Invalid member ID was specified (must be an integer)\n");
  if (!check_pass ($dbh, $member_id, $password)
      && !check_pass ($dbh, 0, $password))
    die ("Invalid password\n");

  # Examine the metadata for the member table to determine whether
  # each column allows NULL values. (Make sure nullability is
  # retrieved in uppercase.)

  $stmt = "SELECT COLUMN_NAME, UPPER(IS_NULLABLE)
           FROM INFORMATION_SCHEMA.COLUMNS
           WHERE TABLE_SCHEMA = ? AND TABLE_NAME = ?";
  $sth = $dbh->prepare ($stmt);
  $sth->execute (array ("sampdb", "member"));
  $nullable = array ();
  while ($info = $sth->fetch ())
    $nullable[$info[0]] = ($info[1] == "YES");

  # Iterate through each field in the form, using the values to
  # construct an UPDATE statement that contains placeholders, and
  # the array of data values to bind to the placeholders.

  $stmt = "UPDATE member ";
  $delim = "SET";
  $params = array ();
  foreach ($row as $col_name => $val)
  {
    $stmt .= "$delim $col_name=?";
    $delim = ",";
    # if a form value is empty, update the corresponding column value
    # with NULL if the column is nullable.  This prevents trying to
    # put an empty string into the expiration date column when it
    # should be NULL, for example.
```

9

```
      $val = trim ($val);
      if (empty ($val))
      {
        if ($nullable[$col_name])
          $params[] = NULL; # enter NULL
        else
          $params[] = "";   # enter empty string
      }
      else
        $params[] = $val;
    }
    $stmt .= " WHERE member_id = ?";
    $params[] = $member_id;

    $sth = $dbh->prepare ($stmt);
    $sth->execute ($params);
    printf ("<br /><a href=\"%s\">Edit another member record</a>\n",
            script_name ());
  }
```

这个函数先要再次检查口令以预防这是别人发来欺骗我们的假表单，然后再更新会员记录。这个更新操作需要一些前期处理，如果用户提交的表单里有空白字段，你可能需要把这些空字符串转换为 NULL 值。expiration 数据列就是这样的例子。比如说，假设研究会秘书以管理员口令（因而失效日期是可编辑的）登录，并把字段清空，表明是终身会员是以其会员资格失效日期为 NULL 值（而非空字符串，不是有效日期）为标志的。因此，当提交回来的表单里有空白字段时，我们判断哪列可以为 NULL 值，再插入 NULL（而非空字符串）。

为解决这一问题，update_entry()函数检索 member 数据表的元数据并构造出一个关联数组：各数组元素的下标是各数据列的名字，表明了该数据列是否允许使用 NULL 值。这些信息可从 INFORMATION_SCHEMA 数据库的 COLOMNS 表中获取。需要从表中获取的这些值是数据列名称和它是否可为 NULL 值（即 COLUMN_NAME 和 IS_NULLABLE 值）。

到这里，edit_member.php 脚本就全部完成了。把这个脚本安装到 Web 文档树的 ushl 目录，让会员们知道他们的口令，他们就能通过 Web 在线修改自己的个人资料了。

Part 3 第三部分

MySQL 的系统管理

本部分内容

第 10 章

MySQL 系统管理简介

随着能力的增强，MySQL 的复杂性有所提高。在各种数据库系统中，MySQL 是相对容易使用的，MySQL 的安装和使用不是很难。MySQL 的简易性会促使它的推广，特别在不是、也不想是系统管理员的人群中。它有助于使你成为熟练的计算机专业人员，不过这不是成功运行 MySQL 的必要条件。

然而，不管你的专业水平如何，MySQL 不会自己运行。必须有人管理和控制它，保证它平稳有效地运行，而且必须有人知道当问题发生时该做些什么。如果你的工作需要你保证 MySQL 良好运行，请继续往下读。

本书的第三部分讨论 MySQL 管理员的各项职责。本章简要说明了管理 MySQL 安装所涉及的职责，随后几章提供了执行该职责的说明。

如果你是一个新的或没有经验的 MySQL 数据库系统管理员，不要被本章中提出的一长串职责吓倒。下面几节中列出的每个任务都很重要，但并不要求你立刻全部学会。如果你喜欢，可以把本书这部分的各章用作参考，当你需要它们时可查询相关题目。

如果你有管理其他数据库系统的经验，就会发现 MySQL 与它们在很多方面都是类似的，你的经验将是一项宝贵的财富。但 MySQL 管理工作有其自己专门的要求，本书的这一部分将帮助你熟悉这些内容。

10.1 MySQL 组件

MySQL 数据库系统是由若干个组件构成的。你应该熟悉这些组件及其用途，以便了解你所管理的系统的性质以及帮助你工作的工具。如果花些时间来了解你要监管的工作，那么，你的工作会更容易一些。因此，你应该懂得 MySQL 的下列各个方面。

MySQL 服务器。MySQL 软件的服务器端主程序 `mysqld` 是每一个 MySQL 数据库系统的核心，对数据库和数据表的所有操作都要通过它来完成。在 Unix 系统上，有几个相关的脚本可以帮助启动 MySQL 服务器运行。`mysqld_safe` 是一个用来启动 MySQL 服务器、监控它并在它意外停机时重新启动它的相关程序。`mysql.server` 脚本在使用运行级子目录来启动各项系统服务的 Unix 版本上很有用。如果打算在同一台主机上运行多个 MySQL 服务器，`mysqld_multi` 脚本可以帮助你更好地管理它们。在 Windows 系统上，可以选择从命令行启动 MySQL 服务器或是把它运行为一项 Windows 服务。

MySQL 客户程序和实用工具程序。有几个 MySQL 程序可以帮助我们与服务器进行交流。就系统管理任务而言，几个最为重要的 MySQL 程序如下所示。

- mysql。一个用来向服务器发送SQL语句和查看结果的交互式程序。mysql程序还可以用来执行批脚本（内容为SQL语句的文本文件）。
- mysqladmin。这个系统管理程序可以用来完成许多任务，包括关停服务器、检查它的配置、在它运行不正常时监控它的工作状态等。
- mysqldump和mysqlhotcopy。用来备份数据库或是把数据库复制到另一个服务器的工具。
- mysqlcheck 和 myisamchk。帮助完成数据库检查、分析、优化以及对受损数据表进行修复的程序。mysqlcheck 适用于 MyISAM 数据表，并在一定程度上也可以用于其他存储引擎所创建的数据表。myisamchk 只适用于 MyISAM 数据表。

服务器语言，SQL。你应该可以使用服务器自己的语言和它通话。例如，你可能需要找出用户权限为什么不按你希望的工作方式进行工作，如果没有替代者能够进入和服务器直接通信，则可使用 mysql 客户程序来进行，使其发出 SQL 查询来检查授权表。

如果你还不知道任何 SQL，至少要对其有基本了解。不具备使用 SQL 的能力会妨碍你的管理工作，而花一点时间来学习则会得到更多的回报。真正掌握 SQL 要花一些时间，但基本技能可以很快得到。对于 SQL 指令以及 mysql 命令行客户程序的使用可参见第 1 章。

MySQL 数据目录。数据目录是服务器存储数据库和状态文件的地方。了解数据目录的结构和内容很重要，你会懂得服务器是如何使用文件系统来表示数据库和数据表的，以及服务器日志放置在何处，其中放置了什么。当你发现放置数据目录的文件系统太满时，你应知道管理整个文件系统磁盘空间分配的选项。

10.2　常规管理

常规管理主要包括操作 MySQL 服务器程序 mysqld，以及向用户提供对服务器的访问能力。下列任务在履行职责时是最重要的。

服务器启动和关闭。应该知道如何由命令行手动启动和停止服务器，当系统要启动和关停时如何安排其自动启动和关停。服务器毁坏或不能正确起动时该做些什么才能使其重新工作，了解这一点也很重要。

用户账号的维护。应该懂得 MySQL 用户账号和 Unix 或 Windows 登录账号之间的差别。应该知道指定哪个用户可以连接至，服务器，从何处连接，他们能做哪些事，以此来设置 MySQL 账号。你还需要知道如何恢复已忘记的口令。

日志文件的维护。应该了解哪些类型的日志可用，哪种有用，以及执行日志文件维护的时间和方法。日志的轮转和失效对于防止该日志填满文件系统是至关重要的。

服务器的配置和优化。MySQL 服务器有很高的可配置性。可以由系统管理员控制的操作特性包括服务器支持的存储引擎、默认的字符集和它的默认时区。

另一个配置问题是服务器调整。用户都希望服务器运行时处于最佳状态，提高服务器运行速度的权宜之计就是购买更多的内存或更快的磁盘。但是这些强力手段并不能帮你了解服务器工作。你应当知道哪些的参数可以调整服务器的工作，以及如何让它们适应你的应用。在某些站点，查询主要是为了检索；而在其他站点，主要是为了插入和修改。要修改哪些参数，取决于你在自己站点观察到的查询。

多个服务器的管理。在某些情况下，在同一个机器上运行多个服务器是有用的。你可以测试一个新的 MySQL 版本，而同时把你现有的服务器留放在原位，或者通过使各组具有自己的服务器来为不

同组的用户提供更好的隐私（后一种情况和因特网服务提供者尤其相关）。对于这种情况，你应该知道如何同时设置多个安装。

MySQL 软件更新。MySQL 版本不断在更新。你应该知道，如何利用 bug 修复和新特性来保持这些版本是最新的。要知道应在哪种情况下推迟更新，如何在版本的稳定和更新之间进行选择。

10.3　访问控制与安全性

当你运行 MySQL 时，确保用户托付给数据库的信息保持安全非常重要。MySQL 管理员负责控制对数据目录及服务器的访问，应懂得下列事项。

文件系统的安全性。Unix 计算机可以接纳几个没有 MySQL 相关管理任务的用户账号。重要的是要保证这些账号不能访问数据目录。这可防止由于复制、删除数据库数据表或由于能读取可能包含敏感信息的日志而损害文件系统级上的数据。你应该知道，如何设置运行 MySQL 服务器要用的 Unix 用户账号，如何来设置数据目录使得用户能占用它，以及如何使用用户权限来启动该服务器运行。

MySQL 服务器安全性。必须懂得 MySQL 安全系统是如何工作的，以便你设置用户账号时为其授予合适的 MySQL 服务器访问权限。通过网络连接至服务器的用户只被允许做他们能做的工作。你不要由于错误理解安全系统而不小心授权用户过高的访问权限！

10.4　数据库的维护、备份和复制

MySQL 数据库管理员都不希望遇上错误百出或者被破坏的数据表。但希望并不能阻止意外的发生。你必须采取措施降低这种风险，同时还要知道在意外发生时应该如何应对。

预防性维护。为了把数据库出现故障或遭到破坏的可能性降到最低，应该提前制定出一套预防性的维护措施。还应该备份数据库，但预防性维护可以减少你求助于那些备份的机会。

数据库备份。万一发生严重的系统崩溃事件，数据库备份将发挥关键的作用。在崩溃发生后，谁都希望数据损失尽可能小，把数据库最大限度地恢复到正常状态。请注意，对数据库备份不同于对系统备份（比如在 Unix 系统上使用 dump 程序备份）。在进行系统备份的时候，数据表对应的文件很可能因为服务器活动而仍处于变化状态，所以恢复那些文件不一定能把数据表里的数据恢复到足够完备的程度。在数据库的恢复工作中，`mysqldump` 程序生成的备份文件更有用，这个程序可以让我们在无需关闭服务器的情况下创建备份。此外，如果需要把数据库迁移到另一个地方（因为现有的硬盘已经满了），也要用到备份文件。

崩溃恢复。如果你已经尽了最大努力但意外仍然发生了，你必须知道如何修复或恢复数据表。崩溃恢复事件极少发生，可一旦发生，就意味着一项痛苦而又高度紧张的工作（尤其是在你正忙得不可开交而电话响、门铃也响的时候）。可要想让用户满意，就必须把事情做好。你必须熟练掌握对 MySQL 数据表进行检查和修复的程序，必须知道如何利用备份文件来恢复受损数据，必须知道如何利用二进制日志来恢复在最近一次备份后又发生的数据修改操作。

数据库迁移。如果你打算把现有的 MySQL 迁移到一台速度更快的主机上运行，你将需要把数据库复制到另外一台机器上去。你应该熟悉完成这种迁移工作的流程。因为数据库文件的具体内容可能与你使用的机器有关，所以简单地把它们从一个系统复制到另一个系统的做法不一定能成功。

数据库复制（replication）。为数据库制作一个备份或副本只相当于在某个特定的时间点对它拍照。还有一种更好的办法是为它建立副本，也就是让两个数据库服务器相互合作，只要它们当中的某个服

务器负责管理的数据库发生了变化，由另一个服务器负责管理的相应的数据库也将发生同样的变化。

要想复制，就必须知道如何把服务器设置为主服务器，如何设置从服务器来同步复制主服务器。万一因为某种原因复制中止，你必须知道如何查找问题的根源并让复制工作重新恢复正常。

以上各节对 MySQL 系统管理员的几大类职责做了简要的描述。接下来的几章将对它们进行详细的讨论。为帮助大家切实可行地履行这些职责，我们还将提供一些关于操作流程和步骤方面的建议。首先讨论 MySQL 的数据目录，它是要管理的主要资源，必须熟悉它的结构和内容。从那里开始，将依次对常规管理任务、MySQL 的安防系统以及数据库的维护和备份工作展开讨论。

第 11 章

MySQL 的数据目录

在概念上，多数相关的数据库系统在很大程度上是类似的。它们管理一组数据库，而且每个数据库包括一组数据表。但是每个系统都有其自己的方式来组织所管理的数据，MySQL 也不例外。默认时，MySQL 服务器 mysqld 管理的所有信息都存储在称作 MySQL 数据目录的位置。所有数据库以及提供服务器工作信息的状态文件和日志就存放于此。如果你有管理 MySQL 设置的职责，熟悉其结构及数据目录的使用对于执行任务是基础。即使你不履行任何 MySQL 管理职责，阅读本章也能获得好处。千万不要放弃使服务器工作得更好的想法。本章主题如下所示。

- **如何确定数据目录的位置**。MySQL服务器上的各种操作都是围绕其数据目录而进行的，必须知道如何确定它的位置才能有效地对它的内容进行管理。
- **服务器如何组织，如何对它管理的数据库及数据表提供访问**。这对于设置预防性维护的计划及多数据表产生错误时进行修复是很重要的。
- **服务器生成什么样的状态文件和日志，以及它们包括的内容**。其内容提供服务器如何执行的有用信息，这在遇到问题时很有用。
- **如何改变数据库目录的默认位置或组织**。这对于管理系统上磁盘资源的分配很重要。例如，平衡跨越驱动器的磁盘活动或把数据重新定位至具有更多空间的文件系统上。也可以使用该知识规划新数据库的位置。

对于 Unix 系统，本章认定存在着用于执行 MySQL 管理任务和运行服务器的登录账号。在本书中，该账号的用户和组名都是 mysql。12.2.1 节下第 1 小节讨论了为 MySQL 管理使用一个指定的登录账号的理由。

11.1 数据目录的位置

默认的数据库目录位置已编译入服务器中。在 Unix 系统上，如果从源代码发行版本安装 MySQL，常用的默认值是/usr/local/mysql/var，如果从二进制发行版本安装，则默认值是/usr/local/mysql/data。如果从 RPM 文件安装，则默认值为/var/lib/mysql。在 Windows 系统上，默认的数据库目录通常是 C:\Program Files\MySQL\MySQL Server 5.0\data 或 C:\mysql\data。

如果从源代码开始编译 MySQL 软件，可以在运行 configure 工具的时候使用--localstatedir = dir_name 命令行选项为数据目录另行指定一个默认的位置。

启动服务器时，使用--datadir=dir_name 选项，可以指定数据库目录的位置。这在你想要把目录放置在默认位置以外的某个地方时是有用的。另一种指定位置的方法是把该位置列在一个选项文件

中，让服务器在启动时读取它。每次启动服务器时你不必把它包括在命令行上。

作为一名 MySQL 系统管理员，应该知道你服务器的数据目录在什么地方，但万一不知道的话（也许你刚和前任交接了工作，而他没有留下什么有用的资料），有好几个办法可以找到它。先介绍一个可以在服务器没运行时使用的办法，然后再介绍一个可以在它正在运行时使用的办法。

查看一下服务器在启动时读取的选项文件。比如说，如果在 Unix 系统上查看/etc/my.cnf 文件或是在 Windows 系统上查看 C:\my.ini 文件，在这个文件的[mysqld]选项组里就应该看到一个 datadir 行：

```
[mysqld]
datadir=/path/to/data/directory
```

这行代码所设置的路径名指向数据目录的位置。

如果不清楚服务器会去什么地方寻找选项文件，可以用如下所示的命令启动它并查看它给出的帮助消息，选项文件的存放位置应该列在开头部分：

```
% mysqld --verbose --help
```

如果服务器正在运行，可以连接它并向它询问数据目录的位置。MySQL 服务器维护着一些与其操作状态相关的系统变量，并且可以随时向你报告它们当中任何一个值。数据目录的位置保存在 datadir 变量里，可以使用 SHOW VARIABLES 语句或是 mysqladmin variables 命令来获得它。下面是用来查看该变量值的 SHOW VARIABLES 语句：

```
mysql> SHOW VARIABLES LIKE 'datadir';
+---------------+-----------------------+
| Variable_name | Value                 |
+---------------+-----------------------+
| datadir       | /usr/local/mysql/var/ |
+---------------+-----------------------+
```

从命令行中，可以使用 mysqladmin。在 Unix 上的输出如下所示：

```
% mysqladmin variables
+---------------+-----------------------+
| Variable_name | Value                 |
+---------------+-----------------------+
...
| datadir       | /usr/local/mysql/var/ |
...
```

在 Windows 上，输出如下所示：

```
C:\> mysqladmin variables
+---------------+-----------------------------------------------+
| Variable_name | Value                                         |
+---------------+-----------------------------------------------+
...
| datadir       | c:\Program Files\MySQL\MySQL Server 5.0\data\ |
...
```

如果有多个服务器在同时运行，它们将在不同的网络接口上监听（TCP/IP 端口、Unix 套接字文件、Windows 命名管道或共享内存）。可以使用相应的连接参数选项依次连接每一个服务器以获得关于数据目录的信息。

如果你想把一个已经创建在某个位置的数据目录转移到另外一个地方，请参阅 11.3 节。

11.2　数据目录的层次结构

MySQL 数据目录包括服务器管理的所有数据库。一般来讲，其组织成一个树结构，利用 Unix 或 Windows 文件系统的分层结构以简单明了的形式。

- 每个数据库都有数据目录下的一个数据库目录。
- 数据库内的数据表、视图和触发器对应于该数据库目录中的文件。

基于子目录和文件的树状数据库实现并非唯一的选择，一种给定的存储引擎完全可以使用一种有别于这种常见形式的存储结构。InnoDB 数据表处理程序把所有数据库中的全部 InnoDB 数据表存放在一个公共表空间内。该表空间是通过一个或多个大文件来实现的，这些文件被当做表示数据表和索引的一个统一的数据结构来处理。默认情况下，InnoDB 将表空间文件存储在数据目录中。

数据目录也可以包括其他文件。

- 服务器进程ID（PID）文件。当其起动时，服务器把其进程ID写入该文件，以便其他程序需要把信号发送该文件时发现其值。（嵌入式服务器不使用该文件。）
- 服务器生成状态文件和日志文件。这些文件提供服务器运行的重要信息，对于管理员很有价值，特别是当出现问题，你想确定问题的原因时。例如，当某些特殊查询伤害服务器时，就能通过检查日志文件来鉴别这个违规的查询。（如果配置服务器时把日志信息写入数据表而不是写入文件，日志数据表将出现在mysql数据库里。）
- 与服务器本身有关的文件，如DES密钥文件、服务器的SSL证书和密钥文件等。系统管理员使用数据目录作为这些文件的存放地点是很常见的。

11.2.1　MySQL服务器如何提供对数据的访问

在通常的客户程序/服务器程序启动过程中使用 MySQL 时，数据目录下的所有数据库均由一个实体——MySQL 服务器 `mysqld`——管理。客户程序从不直接操作数据，而是由服务器提供一个单点联系用于访问数据库，该点用作客户程序和想要使用的数据之间的中介。图 11-1 展示了这种结构。

当服务器起动时，它打开任何一个你请求维护的日志文件，然后通过监听各种网络连接提供一个通向数据目录的网络接口。（12.3 节介绍了选择使用哪个网络接口的细节。）为了访问数据，客户程序建立了一个与服务器之间的连接。然后用 SQL 语句请求通信，以便执行所需的操作，如创建一个数据表，选择记录或修改记录。服务器执行每个操作，并把结果返回给客户。服务器是多线程连接，能够同时为多个客户连接服务。然而，因为修改操作每次只能执行一个，实际效果是串行请求，所以两个客户决不会在同一时间改变一个给定的数据行。

运行使用嵌入式服务器的应用程序时，采用的体系结构稍有不同，因为仅有一个“客户”，服务器连接至它。在这种情况下，服务器监听内部的通信通道，而不是监听网络接口。但即便如此，这类应用程序对数据目录的访问也是在嵌入式服务器的管理和控制之下进行的。如果应用程序对它的嵌入式服务器同时建立了好几个连接的话，就仍有必要对通过这几个连接到达的 SQL 语句的操作行为进行协调。

在正常情况下，如果有服务器用作数据库访问的专用仲裁器，这就为多个进程在同一时间访问数据表可能造成的各种故障提供了预防保障。然而，管理员应该知道，还有许多时候服务器不负责专门控制数据目录。

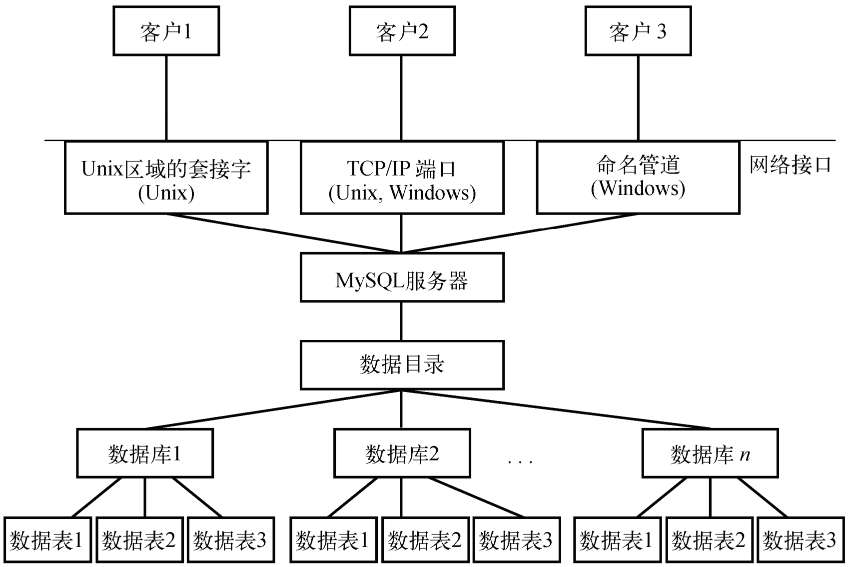


图 11-1 MySQL 服务器控制对数据目录访问的方法

- ❑ 当在一个数据目录上运行多个服务器时。通常一个服务器管理主机上的全部数据库，也有可能运行多个服务器。如果每个服务器只管理自己的数据目录，就没有交互作用的问题。然而，有可能启动多个服务器，并指向同一个数据目录。一般来讲，这并不是一个好主意，不推荐这么做。如果你想试一下，就需要很好地保证你的系统能提供良好的文件锁定，否则，服务器将不能正常工作。如果你允许多个服务器同时往日志文件里面写东西，你的日志文件就可能变成一个混乱之源（而不是帮助信息之源）。
- ❑ 当使用直接访问的维护工具时。例如`myisamchk`和`isamchk`程序用于MyISAM数据表维护、故障诊断、修复和压缩操作，这些程序在对应于该数据表的文件上直接操作。因为工具程序会改变数据表内容，利用它们在服务器工作时在数据表上操作，可能会引起数据表损坏。避免这类问题的最好方法是在运行这些表工具程序时阻止服务器访问该数据表。如果客观条件不允许那么做，就必须知道如何告诉服务器在你使用这类工具直接操作数据表文件时不要去访问该数据表。14.2节给出了一些关于如何在使用这些程序时与服务器进行协调的指导意见。还有一个办法是不要使用`myisamchk`程序，改用`CHECK TABLE`和`REPAIR TABLE`等语句（或`mysqlcheck`程序，它可以替你发出这些语句）。这几条语句是通过服务器实际完成数据表维护操作的，因而不会影响到服务器的正常运行。

11.2.2 MySQL数据库在文件系统里是如何表示的

MySQL 服务器管理的每个数据库都有其自己的数据库目录，它使用它所表示的数据库的名称以数据目录的子目录出现。例如，数据库 mydb 对应于 Unix 或 Windows 上的数据库目录 DATADIR/mydb。

`SHOW DATABASES` 列出了位于数据目录中的目录名。

`CREATE DATABASE db_name` 语句将在数据目录下创建一个名为 db_name 的子目录，作为数据库目录。它还将在这个数据库目录里创建一个 `db.opt` 文件，里面列出了这个数据库默认使用的字符集

和排序方式。在 Unix 系统上，数据库目录的属主是用来运行服务器的那个登录账户，并且只允许该账户访问。

DROP DATABASE 语句的实现几乎同样简单。DROP DATABASE db_name 语句将删除数据目录中的 db_name 子目录，以及该子目录里用来保存数据表和其他数据库对象（视图、触发器等）的文件。这和使用文件系统级命令（如 Unix 系统的 rm 命令或 Windows 系统的 del 命令）手动删除这个数据库目录的效果几乎完全一样。DROP DATABASE 语句与文件系统级命令的区别在于下面几点。

- DROP DATABASE语句只删除用来保存数据表和其他数据库对象的文件（通过文件的扩展名来判断）。如果在数据库目录里还创建了其他的文件或子目录，服务器是不会删除它们的。此时，数据库目录无法删除，DROP DATABASE语句将报告一个错误。这种情况的后果之一是该数据库的名字仍会出现在SHOW DATABASE语句的输出结果里。解决这个问题的办法是先把多余的文件和子目录删掉或是移至别处，然后再次发出DROP DATABASE语句。
- 用删除数据库目录的办法不能安全地删除数据库里的InnoDB数据表。InnoDB存储引擎会为每一个InnoDB数据表在其共享表空间里生成并维护一个数据字典项，InnoDB数据表的内容通常也保存在那里。如果打算删除的数据库里有InnoDB数据表，就必须使用DROP DATABASE语句去删除它，这样才能确保InnoDB存储引擎将更新它的数据字典并从表空间里删除该数据表的内容。

11.2.3 数据表在文件系统里的表示方式

MySQL 支持的存储引擎有 MyISAM、MERGE、MEMORY、InnODB、Falcon、CSV、EDERATED 等许多种。在硬盘上，每个 MySQL 数据表至少对应着一个.frm 格式文件，该文件包含着对数据表结构的描述。.frm 文件由服务器负责创建，相应的存储引擎会再创建一些文件来保存数据行和索引信息。这些文件的名字和结构会根据存储引擎的不同而变化。

在接下来的讨论里，我们将对几种有代表性的存储引擎的特点进行描述，介绍它们如何在硬盘上保存文件。关于这些存储引擎在功能和行为方面有何差异的更多信息，见 2.6.1 节。

MyISAM 存储引擎是 MySQL 服务器默认使用的存储引擎。MySQL 会为每个 MyISAMA 数据表在包含它的那个数据库的数据库目录里创建 3 个文件。这 3 个文件的基本名是一样的，都是数据表名字，扩展名则根据其具体用途而不同。比如说，如果某个 MyISAM 数据表的名字是 mytbl，与之对应的 3 个文件如下。

- mytbl.frm。格式文件，其内容是对该数据表结构的描述。
- mytbl.MYD。数据文件，用来保存该数据表的数据行的内容。
- mytbl.MYI。索引文件，用来保存该数据表的所有索引信息。

MERGE 数据表是一种逻辑构造。它代表着由多个结构完全相同的 MyISAM 数据表组成的一个大数据表。在数据库子目录里，一个名为 mytbl 的 MERGE 数据表对应着以下两个文件。

- mytbl.frm。格式文件。
- mytbl.MRG。这是一个文本文件，里面列出了构成该数据表的MyISAM数据表的名字，每个名字单独占用一行。

MEMORY 数据表是一种驻留在内存里的数据表。每个 MEMORY 数据表在数据库目录里只有一个描述其格式的.frm 文件。除此之外，文件系统里就再也没有其他与之对应的东西了，而这是因为服务器把 MEMORY 数据表里的数据和索引全都存放在内存里而不是存放在硬盘上。当服务器关机时，

MEMORY 数据表的内容将全部丢失。在服务器重新启动后，这个数据表仍然存在（因为它的.frm 文件还存在），但它是空白的。

每个 InnoDB 数据表在数据库目录里有一个用来保存其数据表结构的.frm 文件。至于 InnoDB 数据表的内容，有两种表示形式可供选择，它们都基于表空间。

- ❑ 共享表空间。这个表空间由数据目录里的一个或多个大文件构成。这些表空间组件文件共同形成了一个在逻辑上连续不断的存储区域，其长度等于各组件文件长度之和。在默认的情况下，InnoDB存储引擎会把它的数据表保存到这个共享表空间里。这样的InnoDB数据表在数据库目录里只有一个.frm文件与之对应。
- ❑ 独享表空间。可以配置InnoDB存储引擎，让每个数据表单独使用一个表空间。此时，每个InnoDB数据表在数据库目录里将有两个文件与之对应：一个仍是.frm文件，另一个则是用来存放数据表数据和索引的.ibd文件。

共享表空间还有另一个用途。InnoDB 存储引擎在其内部维护着一个数据字典，其内容是关于各 InnoDB 数据表的信息。这个字典必须保存在共享表空间里，所以即使你使用了独享表空间来存放各 InnoDB 数据表的内容，共享表空间也是必不可少的。

每个 Falcon 数据表在数据库目录里有一个.frm 文件保存着它的数据表结构。Falcon 存储引擎把数据表内容保存在表空间文件里。它有 3 种标准的表空间文件，其中之一用来保存用户的数据表。

- ❑ falcon_master.fts。用来存放供内部使用的数据表。
- ❑ falcon_temporary.fts。用来存放各种临时数据表。
- ❑ falcon_user.fts。Falcon存储引擎默认使用这个表空间文件来存放用户的数据表。

Falcon 存储引擎把它的标准表空间文件都创建在数据目录里。它还可以应用户要求创建额外的表空间文件，但这类表空间文件不必创建在数据目录里。

CSV 存储引擎把数据行保存为 CSV（comma-seperared value，意思是“以逗号分隔的值”）格式的普通文本。每个 CSV 数据表在数据库目录里有两个文件：一个是用来保存数据表结构的.frm 文件，另一个是用来保存数据行的.CSV 文件。

FEDERATED 数据表是指向其他 MySLQ 服务器上的某个远程数据表的数据表。换句话说，数据行不是保存在本地而是从远程数据表那里按需检索而来的。因此，没有任何数据或索引被保存在本地。唯一的本地文件是数据库目录里用来保存数据表格式的.frm 文件。

11.2.4　视图和触发器在文件系统里的表示方式

每个视图和触发器对象在包含着该对象的数据库的数据库目录里对应着一个文件。

每个视图包含一个.frm 文件，它包含着该视图的定义和其他相关属性。这个文件的基本名和视图的名字一样，所以一个名为 `myview` 的视图对应着一个名为 myview.frm 的文件。

触发器保存在一个.TRG 文件里，里面包含着触发器的定义和其他相关属性。触发器文件的基本名和触发器所属的那个数据表的名字一样。比如说，如果一个名为 `mytrig` 的触发器与一个名为 `mytbl` 的数据表相关联，这个触发器将保存在 mytbl.TRG 文件而不是 mytrig.TRG 文件里。事实上，在这个文件里可能保存着多个触发器：同一个数据表可以有多个触发器，而服务器将把它们的定义集中保存在同一个.TRG 文件里。

11.2.5　SQL语句与数据表文件操作的对应关系

每种存储引擎使用一个.frm 文件来保存数据表格式（定义），所以 SHOW TABLE FROM db_name 语句的输出结果和 db_name 数据库目录里的.frm 文件的基本名清单是一样的。

在创建 MySQL 所支持的任意类型的数据表时，需要发出一条 CREATE TABLE 语句来定义该数据表的结构，其中就包括用一个 ENGINE = engine_name 子句，表明指定想使用哪种存储引擎。如果省略了 ENGINE 子句，MySQL 将使用其默认存储引擎（即 MyISAM 存储引擎，如果你没有改变过它的话）。服务器将为新数据表创建一个.frm 文件以保存该数据表的定义的内部编码，然后告诉相应的存储引擎去创建与这个数据表相关的其他文件。比如说，MyISAM 存储引擎将创建一个.MYD 数据文件和一个.MYI 索引文件，CSV 存储引擎将创建一个.CSV 数据文件。如果是 InnoDB 数据表，存储引擎将在相应的 InnoDB 表空间里为新数据表创建一个数据字典项并对其数据和索引信息进行初始化。在 Unix 系统上，为新数据表而创建的一切文件的属主和访问模式，将被设置为只允许用来运行 MySQL 服务器的那个登录账户进行访问。

当发出一条 ALTER TABLE 语句时，服务器将重新编码那个数据表的.frm 文件以反映这条语句对其结构的修改，同时还会对数据文件和索引文件的内容做出必要的修改。在发出 CREATE INDEX 和 DROP INDEX 语句时也会发生这种事情，因为 MySQL 在其内部把它们当做 ALTER TABLE 语句来处理。如果某条 ALTER TABLE 语句改变了数据表的存储引擎，数据表常数将被传递给新存储引擎，新存储引擎将把数据表的内容重新写入适当类型的文件里去。

MySQL 通过删除与数据表对应的文件来实现 DROP TABLE 语句。如果删除的是一个 InnoDB 数据表，InnoDB 存储引擎还将更新它的数据字典并释放该数据表在 InnoDB 共享表空间里占用的空间。类似地，Falcon 存储引擎将释放数据表在相应的表空间文件里占用的空间。

对某几种存储引擎如 MyISAM、MERGE 或 CSV 而言，可以通过在数据库目录里删除某个数据表所对应的文件来手动删除该数据表。对另外几种存储引擎如 InnoDB、Falcon 或 MEMORY 而言，因为数据表的某些部分在文件系统里没有与之对应的文件，所以用来删除这几种数据表的 DROP TABLE 语句没有相应的等效文件系统命令。比如说，存储在共享表空间里的 InnoDB 数据表在文件系统里总是有一个.frm 文件与之对应，但删除那个文件并不能彻底删除该数据表。InnoDB 数据字典只能由 InnoDB 存储引擎更新，简单地删除.frm 文件将导致共享表空间里的数据表数据和索引成为无主的“孤儿”。

如果某个 InnoDB 数据表有它自己的独享表空间，它在数据库目录里就会有它自己的.frm 文件和.ibd 文件。但即便如此，删除那些文件仍不能正确地删除这个数据表，因为 InnoDB 存储引擎还是没有机会更新它的数据字典。因此，InnoDB 数据表必须使用 DROP TABLE 语句来删除，这样才能让 InnoDB 存储引擎在删除文件的同时更新其数据字典。

11.2.6　操作系统对数据库对象的命名规则有何影响

MySQL 有它自己的一套规则来命名数据库和其他对象，如数据表。2.2 节对这套规则进行了详细的讨论，这里再简要概括一下。

- 不带引号的标识符只允许使用系统字符集（utf8）里的字母和数字字符以及下划线和美元字符（“_”和“$”）来构成。
- 用反引号括起来的标识符允许包含其他字符（比如说，`odd?name!`）。如果想使用一个SQL保留字作为标识符，也必须用反斜线把它括起来。如果启用了ANSI_QUOTE SQL模式，反斜线

和双引号将都可以用来括住标识符。

- 标识符的最大长度是64个字符。

此外，MySQL 服务器主机的操作系统也可能对 MySQL 标识符有其他一些限制。这类限制源于文件系统的命名规则，因为数据库和数据表的名字对应着子目录和文件的名字。每个数据库在文件系统里被表示为它的数据库目录，不管使用何种存储引擎，每个数据表在文件系统里至少对应着一个.frm 文件。因此，在命名 MySQL 标识符的时候还需要考虑以下因素。

- MySQL允许数据库和数据表的名字长达64个字符，但这个长度不得超过操作系统所允许的最大长度。
- 底层文件系统是否区分大小写会影响到对数据库和数据表的命名和引用。如果文件系统区分大小写（绝大多数Unix操作系统都是如此），文件名abc和ABC将代表着两个不同的文件。如果文件系统不区分大小写（例如Windows，Mac OS X操作系统的HFS+文件系统等），文件名abc和ABC将代表着同一个文件。如果你在一个对文件名区分大小写的服务器上开发了一个数据库，而以后又有可能需要把这个数据库移动或是复制到一个对文件名区分大小写的服务器上的话，就要时刻注意这个问题。

在 MySQL 5.1.6 之前的版本里，MySQL 标识符可能还会受到来自文件系统的其他一些限制。

- 数据库和数据表的名字不得包含在文件名里不允许出现的非法字符。但这些非法字符在不同的操作系统里有不同的规则，这意味着MySQL允许出现在名字里的某些字符最好还是不要使用。比如说，在Unix系统上，可以在一个用引号括起来的标识符里加上一个“*”字符作为某个数据表的名字。但Windows系统不允许文件名里出现“*”字符，如果不先去掉这个字符，就不能把这个数据表复制或移动到Windows系统上。总而言之，最好的办法是坚持使用普通字符，尽量避免使用奇怪的字符。
- 数据库或数据表的名字不得包含路径名分隔符，就算用引号括起来也不行。Unix和Windows系统分别使用“/”和“\”字符作为其路径名分隔符，这两个字符都不允许出现在MySQL标识符里。在任何平台都禁止使用这两个字符，可以让数据库和数据表在这两种平台之间的迁移工作减少许多麻烦。

由于文件名里的非法字符或是不可移植字符而导致的上述问题，在 MySQL 5.1.6 及更高的版本里已得到彻底解决，MySQL 服务器将对 MySQL 标识符里可能会导致非法文件名的特殊字符进行编码。这个编码操作解除了在名字里不允许使用“/”、“\”和其他一些字符的限制。在把一个可以用在 SQL 语句里的名字映射为相应的文件名时，数字和拉丁字母以外的每个字符将被应映射一个“@*xxxx*”形式的字符编码。比如说，“?”和“!”字符的编码是 003f 和 0021，于是数据表名 odd?name!所对应的.frm 文件的名字将是 odd003fname0021.frm。与数据表相关联的其他文件的命名方式与此类似。

11

在从某个老版本升级到 MySQL 5.1.6 或更高版本时，记得要用下面这条命令让服务器对全体数据库和全体数据表的名字进行必要的编码：

```
% mysqlcheck --all-databases --check-upgrade --fix-db-names --fix-table-names
```

正如刚才提到的那样，文件系统是否区分大小写会影响到对数据库和数据表的命名。解决这个问题的办法之一是固定使用一种大小写形式。另一个办法是在启动服务器时把 lower_case_table_names 系统变量设置为 1，如此设置有两个效果。

- 在为某个数据表创建相应的硬盘文件之前，服务器会先把该数据表的名字转换为小写字母。
- 当你在某个语句里引用了这个数据表时，服务器在去硬盘上寻找这个数据表之前会先把它的

名字转换为小写字母。

这两个动作的结果是，无论文件系统是否区分大小写，所有的名字都将不区分大小写。这样的话，在系统间移动数据库和表将更容易。但要特别提醒大家注意的是，如果打算使用这个策略，就必须在开始创建任何一个数据库或数据表之前（而不是之后）去配置服务器以设置 lower_case_table_names 系统变量。如果在设置这个变量之前已经创建了一些在名字里有大写字母的数据库或数据表，这项设置将不会产生预期的效果，因为硬盘上已经保存着一些不全是小写字母的名字了。避免这个问题的办法很简单，先把名字里有大写字母的数据表全部重新命名为小写字母的形式，然后再去设置 lower_case_table_names 系统变量。（ALTER TABLE 或 RENAME TABLE 语句都可以用来重新命名数据表。）如果有非常多的数据表需要重新命名，或者还有一些数据库的名字里也有大写字母，那么更简单的办法是先转储这些数据库，等设置好 lower_case_table_names 系统变量之后再把重新创建。

(1) 用 mysqldump 工具依次转储各数据库：

```
% mysqldump --database db_name > db_name.sql
```

(2) 用 DROP DATABASE 语句删除那些数据库。

(3) 关停 MySQL 服务器，重新配置它以设置 lower_case_table_names 系统变量，然后重新启动服务器。

(4) 用 mysql 程序重新加载转储文件：

```
% mysql < db_name.sql
```

因为已经设置了 lower_case_table_names 系统变量，为各数据库和数据表而创建的硬盘文件将有一个全部是小写字母的名字。

lower_case_table_names 系统变量还有另外几种可取值，更多信息请参阅附录 D。

无论 lower_case_table_names 系统变量的设置情况是怎样的，Falcon 存储引擎都将以不区分大小写的方式处理数据库和数据表的名字。

11.2.7　影响数据表最大长度的因素

MySQL 中的数据表长度是有界限的，但会受到很多因素的影响，所以想精准地确定这些界限并不是件简单的事情。

首先，操作系统对单个文件的最大长度有一个限制。低至 2GB 的长度上限很常见，但因操作系统长期以来放松对文件长度的限制，这种情况已近乎绝迹。操作系统对文件长度的限制同样适用于数据表所对应的那些文件，比如 MyISAM 数据表所对应的.MYD 和.MYI 文件。它还适用于构成 InnoDB 表空间的那些文件。不过，InnoDB 表空间的总长度可以轻易超过单个文件的最大长度，只要把它配置成使用多个文件、每个文件的长度又都是最大值就行了。避免文件长度限制的另一个办法是在 InnoDB 表空间里使用硬盘的原始分区。硬盘原始分区里的表空间组件可以和硬盘分区本身一样大。InnoDB 存储引擎在这方面的配置步骤见 12.7.3 节的第 1 小节。

除了操作系统方面的限制，MySQL 对于数据表的长度也有它自己的内部限制，因存储引擎而异。

- 对于MyISAM数据表，单个.MYD或.MYI文件的默认最大长度是256TB。但在创建数据表的时候，可以利用AVG_ROW_LENGTH和MAX_ROWS选项把单个文件的长度上限加大到65 536TB。（请参见附录E里对CREATE TABLE语句的说明。）这些选项会影响到MySQL内部的数据行指针宽

度，这个宽度用来确定数据表所能容纳的数据行的最大个数。当某个MyISAM数据表增长到接近其最大长度并报告135或136号错误时，我们可以利用`ALTER TABLE`语句来加大这两个选项的值。如果想直接修改默认的MyISAM指针宽度，设置`myisam_pointer_size`系统变量即可，新设置将对那以后创建的数据表生效。

- MERGE数据表的最大长度是其成员MyISAM数据表的最大长度的总和。
- 对于InnoDB数据表，InnoDB共享表空间的最大长度是40亿个页面，默认的页面长度是16 KB。(如果从源代码开始重编译MySQL，就可以在8 KB到64 KB的范围内选用InnoDB页面的长度。)表空间的最大长度也是存储在表空间里的单个InnoDB数据表的长度上限。如果把InnoDB存储引擎配置成使用独享表空间的情况，每个InnoDB数据表的内容将存储在它自己的.ibd文件里。此时，InnoDB数据表的长度将受限于操作系统的文件长度上限。
- Falcon表空间文件的最大长度是128TB，Falcon数据表里的数据行的个数最多不能超过2^{32}个。

对于把数据和索引分别存放在不同文件里的存储引擎，只要那些文件当中有一个到达文件长度上限，也就到达了该数据表的长度上限。对于一个 MyISAM 数据表，索引的情况对哪一个文件最先到达文件长度上限有很大的影响。如果这个数据表没有或只有很少的索引，一般是数据文件最先到达上限。如果它是一个索引众多的数据表，索引文件可能会先到达上限。

`AUTO_INCREMENT` 列的出现会限制数据表可能具有的行数。例如，如果该列数是 `TINYINT UNSIGNED`，其最大值可以为 255，则数据表可能具有的最大行数也为 255。较大的整数类型允许有更多的行数。一般来说，在数据表上放置 PRIMARY KEY 或 UNIQUE 索引会将其行数限制在索引中特定数的最大值之内。

为了确定最大的实际数据表尺寸，必须考虑所有可能的因素。有效的最大数据表尺寸可能由其中的最小因素确定。假如你要创建一个 MyISAM 表，如果使用默认的数据指针大小，MySQL 将允许数据和索引文件达到 256TB。但如果操作系统将文件大小限制为 2GB，那么数据表文件的有效限值也是 2GB。另一方面，如果你的系统支持大于 4GB 的文件，那么确定数据表大小的因素将是 MySQL 内部数据指针的大小，这是你可以控制的。

InnoDB 数据表安装在共享表空间内。一个 InnoDB 数据表可以和表空间一样大表空间可以跨越多个文件，以便空间更大。但是很可能有许多 InnoDB 数据表共用相同的区域，这样的话，每个数据表不仅要受到表空间尺寸的限制，还要受到有多少数据表空间分配给其他数据表的限制。只要表空间不满，任何 InnoDB 数据表都可以增长。相反，当表空间填满时，InnoDB 数据表不能再增长，除非增加另外的组成部分使表空间加大。也可以使最后的表空间部分自动扩展，只要超过你系统文件尺寸的界限，而且还有磁盘空间，该数据表空间就能增长。请参见 12.7.3 节下的第 1 小节了解表空间配置的讨论。

Falcon 表空间文件将从它们的初始长度开始自动增长，但长度上限是 128TB。如果某个表空间变得过于“拥挤”，可以另行创建一个表空间并把数据表从某一个表空间移动到另外一个。12.7.4 节对 Falcon 表空间的创建情况进行了讨论。

11.2.8 数据目录的结构对系统性能的影响

MySQL 数据目录的结构容易理解，因为它以一种自然的方式使用文件系统的分层结构。同时，该结构具有一定的隐含性能，特别是与打开表示数据库数据表的文件的操作有关的性能。

采用这种数据目录结构的后果之一是，对于那些把每个数据表存放在它们自己的文件里的存储引

擎而言，每打开一个数据表，就至少需要用掉一个文件描述符。如果某个数据表对应着多个文件，打开这个数据表将需要多个文件描述字符而不仅仅是一个。MySQL 服务器在文件描述符的缓存表现得相当聪明，但在一个繁忙的服务器上，为众多客户同时提供服务或是执行一条涉及多个数据表的复杂语句都需要大量的文件描述符。这很有可能成为一个瓶颈，因为文件描述符在许多系统上都属于一种珍稀资源，尤其是在那些把默认分配给每个进程的文件描述符的最大个数设置得相当低的系统上。如果某种操作系统在这方面设置了一个很低的上限值并且没有提供加大这个数值的手段，就不适合用来承载一个繁忙的 MySQL 服务器。

用自己的文件表示每个数据表的另一个效果，是数据表的打开时间随着数据表数量的增加而增加。打开数据表的操作与操作系统提供的文件打开操作相映射，这样就受到了系统目录查阅程序的效率的限制。通常这没有多大问题，但是当你需要大量数据库内的数据表时就得考虑某些事情。例如，一个 MyISAM 数据表用 3 个文件来表示。如果你想有 10 000 个 MyISAM 数据表，则数据库目录将含有 30 000 文件。有这么多的文件，你就得注意，文件打开操作需要花费时间，因而慢下来。如果牵涉到这个问题，你得使用一种能高效处理大量文件的文件系统。例如，即使有大量的小文件，XFS 和 JFS 仍表现出良好的性能。如果不能另外一个文件系统，则必须根据应用程序的需要重新考虑数据表的结构，并且重新组织数据表。自问一下实际是否需要如此多的数据表，有时候应用程序往往不需要这么多数据表。为每个用户都建立一个数据表的应用程序，会产生许多数据表，这些数据表都有相同的结构。如果把这些数据表组合为一个数据表，只要增加另一列来识别每行用户即可。如果这能显著减少数据表数量，应用程序的性能将会得到提高。

一定要在设计数据库时，考虑这种决策对给定的应用程序是否值得。不以这种方式合并数据表的理由如下所述。

- 增加磁盘空间的需求。合并数据表能减少所需数据表的数量（减少数据表打开时间），但增加了另一列（增加磁盘空间需求）。这是典型的时间与空间的权衡，必须确定哪个因素最重要。如果速度是头等的，就得愿意牺牲一点额外的磁盘空间。如果空间要紧，就应使用多个数据表，这就免不了会延时。
- 安全性考虑。这可能会束缚你的能力，或者需要合并数据表。每个用户使用一个数据表的理由是，每个数据表只允许具有数据表级权限的用户进行访问。如果你合并了数据表，所有用户用的数据将在同一个数据表中。

MySQL 不提倡限制用户只能访问特定行。这样，就无法在保证访问控制权的前提下合并数据表。可以通过视图为当前用户选择行，并允许访问。在另一方面，如果应用程序控制对数据的所有访问（用户绝不能直接连接至数据库），你就能合并这些数据表，而且使用应用程序的逻辑对合并结果进行数据行级的强制访问。

创建许多数据表而无须如此多文件的另一种方法是使用 InnoDB 数据表并将它们保存在 InnoDB 共享表空间内。InnoDB 存储引擎只和每个数据表专用的.frm 文件有关，并且把所有 InnoDB 数据表用的数据和索引信息一起存入 InnoDB 共享表空间内。这减少了表示该数据表所需磁盘文件的数量，实际上还大量减少了打开数据表所需文件描述符的数量。InnoDB 对表空间的每个组成文件只需要一个描述符（这在服务器工作生存期间是恒定不变的），简单地说，在读取数据表.frm 文件时，只需打开任一数据表的一个描述符。

对于 Falcon，可使用一种类似的方法，让多个数据表的数据和索引保存一个表空间文件内。

11.2.9 MySQL状态文件和日志文件

除了数据库目录外，MySQL 数据目录含有许多状态文件和日志文件，如表 11-1 所示。这些文件的默认位置是服务器的数据目录，其中许多默认名是从在数据表中表示为 HOSTNAME 的服务器主机名得来的。二进制日志和中继日志都将被创建为一组按顺序编号的文件，用 nnnnn 来表示。下面这份表格只列出了服务器级的状态文件和日志文件。个别存储引擎可能还会创建它们自己的日志或其他文件。比如说，InnoDB 和 Falcon 存储引擎就会这么做。

表 11-1 MySQL 状态文件和日志文件

文件类型	默 认 名	文件内容
进程ID文件	*HOSTNAME*.pid	服务器进程ID
错误日志	*HOSTNAME*.err	启动/关闭事件和错误条件
一般查询日志	*HOSTNAME*.log	连接/断开事件和查询信息
二进制日志	*HOSTNAME*-bin.nnnnnn	修改数据的语句的二进制表示
二进制日志的索引文件	*HOSTNAME*-bin.index	现有二进制日志文件的清单
延迟日志	*HOSTNAME*-relay-bin.nnnnnn	从属服务器从主服务器收到的数据修改信息
延迟日志索引	*HOSTNAME*-relay-bin.index	当前延迟日志文件清单
主服务器信息	master.info	用于连接主服务器的参数
延迟信息	relay-log.info	延迟日志处理的状态
慢查询日志	*HOSTNAME*-slow.log	耗时很长的语句的文本

对于常规查询日志和慢查询日志，可以选择是让服务器把日志信息写入一个日志文件、写入 mysql 数据库里的某个日志数据表，或者这两个地方都写。12.5.6 节对把日志信息写入数据表的有关问题进行了详细的讨论。

1. 进程ID文件

MySQL 服务器会在启动时把它的进程 ID（PID，process ID）写入 PID 文件，在结束运行时又会删除该文件。其他进程可以利用这个文件来确定 MySQL 服务器是否正在运行以及（如果正在运行的话）它的进程 ID。比如说，如果操作系统在系统关机过程中调用了 mysql.server 脚本去结束 MySQL 服务器的运行，该脚本就会查看这个 PID 文件以确定自己应该向哪一个进程发出终止运行信号。

如果服务器无法创建 PID 文件（比如说，如果在光盘之类的只读介质上运行它），它将把一条消息写入出错日志并继续。

嵌入式服务器不使用 PID 文件，或者说它根本不需要 PID 文件，因为嵌入式服务器是由其宿主应用程序来启动和关闭的。

2. MySQL日志

MySQL 能够维护多种类型的日志文件。大部分日志功能都是可选的，可以使用相应的服务器启动选项只启用需要的日志并（如果不喜欢它们的默认名的话）起一个名字。请注意，日志文件有可能增长到非常巨大，所以千万不要让它们把文件系统都占满了。应该定期对日志文件进行失效处理以保证它们所占用的空间总量在限度以内。

本节将对几种常用的日志文件进行介绍。如果想了解更多关于日志、服务器日志行为的控制选项以及如何对日志文件进行失效处理等方面的信息，请参阅 12.5 节。

错误日志（error log）的内容是服务器在系统发生意外状况时生成的诊断信息。如果服务器启动

失败或是意外退出，这个日志就很有用了，因为故障的根源往往就记录在它里面。

常规查询日志（general query log）的内容是关于服务器操作的常规信息：谁在连接服务器，从什么地方连接，发出了什么语句，等等。二进制日志也包含着语句信息，但仅限于那些对数据库的内容做出了修改的语句。在复制机制中的主服务器上，它还包含让从服务器保持同步所需要的时间戳信息。二进制日志的内容是一些以二进制格式记载的“事件”，把这些事件提供给 mysql 客户程序作为输入就可以执行它们。配套的二进制日志索引文件列出了服务器当前正在维护的二进制日志文件。

二进制日志对系统崩溃后的数据库恢复工作有着重要意义，把二进制日志文件馈入服务器执行相当于把上次备份后执行过的数据更新操作重复一遍。这就使我们能够把数据库的状态恢复到崩溃发生前的那一刻。二进制日志在复制机制中也发挥着重要作用，主服务器上的数据更新操作就是通过二进制日志传递到从服务器去的。我们将在第 14 章对备份和复制操作进行更详细的讨论。

下面是常规查询日志的一段样板内容，这段信息对应着一个简短的客户会话，用户在 test 数据库里创建了一个数据表，在该数据表里插入了一个数据行，然后丢弃了该数据表：

```
080412 11:38:34      31 Connect     sampadm@localhost on sampdb
080412 11:38:42      31 Query       CREATE TABLE mytbl (val INT)
080412 11:38:47      31 Query       INSERT INTO mytbl VALUES(1)
080412 11:38:52      31 Query       DROP TABLE mytbl
080412 11:38:56      31 Quit
```

如上所示，常规日志包含有关日期和时间、服务器线程 ID（连接 ID）、事件类别、具体事件信息的数据列。如果某一行的日期和时间字段缺失，它的值将和前面有这个字段的那一行的日期和时间字段值相同。（换句话说，服务器只在前后两个日志记录项的日期和时间字段值不一样的时候才会把后一项的日期和时间字段值记载到日志里。）

下面是用 mysqlbinlog 工具程序去查看同一次会话在二进制日志里留下的日志信息时看到的内容。（由于本书的页面宽度有限，对这段输出内容稍微做了些排版。）在这份输出内容里可以看到，语句末尾的分号结束符都保留着，这使它们可以在数据库恢复操作中直接用作 mysql 程序的输入来重复一遍这些数据的更新操作：

```
# at 1222
#080412 11:38:42 server id 1  log_pos 1222      Query   thread_id=31
exec_time=0     error_code=0
use sampdb;
SET TIMESTAMP=1092328722;
CREATE TABLE mytbl (val INT);
# at 1287
#080412 11:38:47 server id 1  log_pos 1287      Query   thread_id=31
exec_time=0     error_code=0
SET TIMESTAMP=1092328727;
INSERT INTO mytbl VALUES(1);
# at 1351
#080412 11:38:52 server id 1  log_pos 1351      Query   thread_id=31
exec_time=0     error_code=0
SET TIMESTAMP=1092328732;
DROP TABLE mytbl;
```

作为一名系统管理员，应该确保日志文件的安全，不允许非授权用户读取它的内容。这是因为它们也许会有口令之类的敏感信息。比如说，下面这条日志记录项就包含有 root 用户的口令，你肯定

不想让随便什么人都能访问这类信息：

```
080412 16:47:24       44 Query       SET PASSWORD FOR
                                     'root'@'localhost'=PASSWORD('secret')
```

在默认的情况下，服务器将把日志文件写到数据目录里。因此，作为确保日志文件安全的预防措施之一，应该让服务器主机上的数据目录只允许 MySQL 系统管理员所使用的登录账户来访问。相关设置步骤见 13.1.2 节。

11.3 重新安置数据目录的内容

本章的前几部分讨论了默认配置中的数据目录结构，即位于其中的所有数据库、状态文件和日志文件。然而，在确定数据目录内容的放置时还有一些活动余地。你可以重新安置数据目录本身或其中的某些元素。下面是你要这样做的几个理由。

- 如果包含数据目录的文件系统满了，你可以把数据目录放在容量更大的文件系统上。
- 如果数据目录在一个很忙的磁盘上，可以把它放在活动量少的驱动器上，以平衡跨越物理设备的磁盘活动。可以把数据库和日志文件放在不同的驱动器上，或者以同样的理由把数据库分布在各驱动器上。同样，InnoDB共享表空间在概念上是一个大的存储块，但可把其各个组成文件放在不同的驱动器上，以改善性能。如果你使用了分区数据表，那么你就可以在各个分区完成这些事。
- 把数据库和日志放在不同的磁盘上，有助于减少单个磁盘故障可能引起的损坏。
- 你需要运行多个服务器，每个都有自己的数据目录。这是解决每个进程文件描述符极限值的问题的一种方法，特别是当你不能重新配置系统内核设置较高的极限值时。

本节的其余部分讨论数据目录的哪一部分可以移动，以及你如何移动。

11.3.1 重新安置工作的具体方法

有两种方法可以重新安置数据目录或其中的元素。

首先，在任何平台上，你都可以在服务器启动时指定一个选项，既可以在命令行上，也可以在选项文件中。例如，如果你要指定数据目录的位置，可以用命令行上的`--datadir=dir_name`选项启动服务器，也可把下列各行放入一个选项文件中：

```
[mysqld]
datadir=dir_name
```

一般来讲，服务器选项中的选项文件组名是`[mysqld]`，如示例中所示。根据你的情况，或许其他选项组合更合适。例如，`[embedded]`组适用于嵌入式服务器。如果你使用`mysqld_multi`运行多个服务器，该组名将是`[mysqldn]`形式，其中`n`是一个整数，和一个特定的服务器实例关联。12.2.3 节讨论了哪个选项组适用于哪种服务器启动方法，还提供了运用多个服务器的说明。

其次，在 Unix 上，你可以移动要重新安置的文件或目录，然后在指向新位置的原来位置处生成一个符号链接。

这些方法并非适用于你能重新安置的所有情况。表 11-2 介绍了哪些是能够重新安置的，以及哪个重新安置方法可以采用。使用一个选项文件，可以在全局性选项文件（例如在 Unix 下为/etc/my.cnf，而在 Windows 下为 C:\my.cnf）中指定选项。

表 11-2　重新安置的各种方法

重新安置实体	适用的重新安置方法
整个数据目录	启动选项或符号链接
各个数据库目录	符号链接
各个数据表	符号链接
InnoDB数据表空间文件	启动选项
服务器PID文件	启动选项
日志文件	启动选项

11.3.2　重新安置注意事项

在重新安置之前，一定要备份数据，以便在重新安置操作混乱时恢复数据。同样在执行任何重新安置操作之前，都应该关停服务器，以后再重新启动。对于某些类型的重新安置而言，例如移动数据库目录，有可能保持服务器运行，但并不推荐。如果你要这样做，一定要保证服务器不访问正移动的数据库。还一定要在移动数据库之前发出 FLUSH TABLES 语句，保证服务器关闭所有打开的数据表文件。不遵守这几点可能会导致数据表损坏。

11.3.3　评估重新安置的效果

无论重新安置什么，首先需要确定这个操作会收到好的效果。例如，在 Unix 上，可以使用 du、df 和 ls-l 命令来获取磁盘空间信息，然而，必须正确理解文件系统的布局。

在对数据目录转移工作进行事前评估的时候，要特别留意一些常见的“陷阱”。我们来看一个这方面的例子。假设数据目录是/usr/local/mysql/data，你想把它转移到/var/mysql，因为 df 命令的输出结果表明/var 文件系统有更多的可用空间：

```
% df -k /usr /var
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda5            28834716K 24078024K  3291968K  88% /usr
/dev/sda6            28834716K  9175456K 18194536K  34% /var
```

先用 du-s 命令了解一下转移数据目录可以在/usr 文件系统上释放多少空间：

```
% du -s /usr/local/mysql/data
3264308K /usr/local/mysql/data
```

这个结果表明，把数据目录从/usr 转移到/var 将可以释放/usr 上约 3GB 空间。果真如此吗？为了查明真相，再用 df 命令去检查数据目录。假设看到了如下所示的输出结果：

```
% df -k /usr/local/mysql/data
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/sda6            28834716K  9175456K 18194536K  34% /var
```

很奇怪，为什么 df 命令报告的是/var 文件系统上的空间使用情况呢？下面这条 ls-l 命令给出了答案：

```
% ls -l /usr/local/mysql/data
lrwxrwxr-x  1 mysql  mysql  10 Dec 11 23:46 data -> /var/mysql
```

这个输出结果告诉我们，/usr/local/mysql/data 是一个指向/var/mysql 的符号链接。换句话说，数据

目录早已被转移到/var 文件系统并被替换为一个指向那里的符号链接。为了释放/usr 的大量空间而把数据目录转移到/var 的故事就到此为止吧！

这个例子的教训是，只要在评估转移效果时多花一点儿时间，就可以让自己少干许多劳而无功的傻事。

11.3.4 重新安置整个数据目录

重新安置数据目录时要关停 MySQL 服务器，并把数据目录移至新位置。然后用明确指示新位置的--datadir 选项重新启动服务器。在 Unix 上，除使用--datadir 外，也可以在原来的数据目录位置创建一个符号链接，指向新位置。

11.3.5 重新安置各个数据库

服务器要想得知数据目录中的数据库目录，唯一的办法是用符号链接方法重新安置数据库。这个过程在 Unix 和 Windows 下有所不同。

在 Unix 下，按下述方法进行。

(1) 关停正在运行的服务器。

(2) 把数据库移至新位置，或复制数据库并删除原来的那个。

(3) 在具有原来数据库名的数据目录中创建一个符号链接，并指向数据库新的位置。

(4) 重新启动服务器。

下面举例展示如何使用这些步骤把数据库 bigdb 从/usr/local/mysql/data 目录移至/var/db：

```
% mysqladmin -p -u root shutdown
Enter password: ******
% cd /usr/local/mysql/data
% tar cf - bigdb | (cd /var/db; tar xf -)
% rm -rf bigdb
% ln -s /var/db/bigdb bigdb
% mysqld_safe &
```

以 MySQL 管理员身份登录时应该执行这些命令。

在 Windows 下，数据库重新安置的处理稍有不同。

(1) 关停正在运行的服务器。

(2) 把数据库目录移至新位置，或者复制它，并删除原来的那个。

(3) 在 MySQL 数据目录中创建一个.sym 文件，用作一个符号链接，让 MySQL 服务器知道在何处找到重新安置的数据库目录。这个文件的基本名就是数据库名。例如，如果把 sampdb 数据库从 C:\mysql\data\sampdb 移至 E:\mysql-book\sampdb，则要创建一个名叫 C:\mysql\data\sampdb.sym 的文件，其中包括下面的内容：

```
E:\mysql-book\sampdb\
```

(4) 要保证重新启动服务器时启用符号链接。Windows 服务器在默认情况下是启用符号链接的，你也可以用命令行上的--symbolic-links 选项做这些工作，也可以把下列各行放在选项文件中：

```
[mysqld]
symbolic-links
```

当你把一个数据库移至另一个文件系统，试图重分配数据库存储器时，请记住，如果你使用的

InnoDB 数据表存储在 InnoDB 共享表空间内，那么那些表的内容并不存在于数据库目录中。对于主要由 InnoDB 数据表组成的数据库来讲，重新安置数据库目录在存储器分配方面的效果较小，只能重新安置它们的.frm 文件，不能重新安置它们的内容。

类似地，Falcon 数据表内容存储在 Falcon 表空间文件内，而非数据库目录中。

11.3.6　重新安置各个数据表

只有在以下条件全部满足时你才能重新安置数据表。

- 你使用的是Unix操作系统，并且你打算重新安置的数据表是一个MyISAM数据表。
- 你的操作系统必须具有一个确实能工作的`realpath()`系统调用。如果确实如此，下面这个查询的结果将是`YES`：

```
mysql> SHOW VARIABLES LIKE 'have_symlink';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| have_symlink  | YES   |
+---------------+-------+
```

只有上述两个条件全部满足时，你才能把数据表的.MYD 数据文件和.MYI 索引文件移到新位置。别忘了在原来的数据文件和索引文件所在的数据库子目录里创建两个符号链接分别指向它们的新位置。（.frm 文件仍需留在原来的数据库子目录里。）此前，你必须在移动文件时，必须停止服务器的运行，或是锁定数据表以防止服务器使用它，具体步骤请见 14.2 节。

11.3.7　重新安置InnoDB共享表空间

在第一次配置 InnoDB 共享表空间时，你需要通过 `innodb_data_home_dir` 和 `innodb_data_file_path` 选项（配置共享表空间的详情参见 12.7.3 节的第 1 节）把它的各组成文件记载到选项文件里。如果你已经建立了数据表空间，就能重新安置它的某些组成文件，例如，把它们分散到不同的文件系统。因为文件的位置是通过开机选项引出的，所以你必须按以下步骤来移动部分或者或全部的表空间文件。

(1) 关停正在运行的服务器。

(2) 移动数据表空间文件或你要重新安置的文件。

(3) 修改定义 InnoDB 配置的选项文件，要反映出你所移动的文件的新位置。

(4) 重新启动该服务器。

11.3.8　重新安置状态文件和日志文件

重新安置 PID 文件或日志文件的步骤是：先关闭 MySQL 服务器，再用一个能够设定文件新位置的选项重新启动它。比如说，如果你想把 PID 文件创建为/tmp/mysql.pid 文件，办法之一是在命令行上使用`--pid-file=/tmp/mysql.pid`选项，办法之二是在某个选项文件里添加以下内容：

```
[mysqld]
pid-file=/tmp/mysql.pid
```

如果 PID 文件名是通过一个绝对路径名给出的，MySQL 服务器将使用该路径名去创建 PID 文件。如果使用相对路径名，MySQL 服务器将在它自己的数据目录中创建这个文件。比如说，如果你给出

的选项是--pid-file= mysqld.pid，PID 文件将被创建为 MySQL 数据目录里的 mysqld.pid 文件。

有些系统会把各种服务器的 PID 文件集中保存在一个特定的子目录（如/var/run）里，而你应该把 MySQL 服务器的 PID 文件也保存到那里，以保持系统操作的一致性。类似地，如果你的系统把日志文件集中保存在/var/log 子目录里，你就应该把 MySQL 的日志文件也放到那里去。不过，有许多系统只允许 root 用户对这些子目录进行写操作，而这意味着你必须以 root 用户的权限来运行 MySQL 服务器，从安全角度讲，这可不是一个好主意。两全其美的办法是：创建两个子目录/var/run/mysql 和/var/log/mysql 并把它们的属主设置为你将用来运行 MySQL 服务器的账户。比如说，如果你打算用来运行 MySQL 服务器的账户的用户名和用户组名都是“mysql”，你就需要以 root 用户的身份执行以下命令：

```
# mkdir /var/run/mysql
# chown mysql /var/run/mysql
# chgrp mysql /var/run/mysql
# chmod u=rwx,go-rwx /var/run/mysql
# mkdir /var/log/mysql
# chown mysql /var/log/mysql
# chgrp mysql /var/log/mysql
# chmod u=rwx,go-rwx /var/log/mysql
```

这样一来，MySQL 服务器将可以毫无问题地在那两个子目录里进行写操作，而你可以在启动它的时候使用特定选项来给出那里的文件。比如说：

```
[mysqld]
pid-file = /var/run/mysql/mysql.pid
log-error = /var/log/mysql/log.err
log = /var/log/mysql/querylog
log-bin = /var/log/mysql/binlog
```

关于与日志文件有关的选项及其用法，请参见 12.5 节。

第 12 章 MySQL 数据库系统的日常管理

如果你是一名 MySQL 系统管理员的话，本章就是为你而准备的。我们将在这一章讨论 MySQL 系统管理员为保证 MySQL 的顺畅运行而必须承担的职责。

- 在安装MySQL之后加强它的安全设置。
- 确保服务器能正常运行尽可能长的时间。
- 创建用户账户，让客户可以访问服务器。
- 维护服务器的日志。
- 为改善性能而修改和监控服务器的操作参数。
- 运行多个服务器。
- 确定是否以及何时为MySQL升级。

其他一些比较重要的系统管理职责将在第 13 章和第 14 章里讨论。

在讨论过程中，本章还将对 MySQL 系统管理员应知应会的几种工具程序做比较详细的介绍，它们是：

- MySQL软件的服务器端主程序mysqld程序；
- 用来启动服务器的mysqld_safe、mysqld.server和mysqld_multi脚本；
- 用来完成各种日常系统管理操作的mysqladmin程序。

本章的大部分内容需要读者熟悉 MySQL 的数据目录才能更好地掌握，它是 MySQL 服务器存放数据库、日志文件和其他信息的场所。关于数据目录的详细讨论请参阅第 11 章。对本章将会用到的 SQL 语句和程序的详细介绍请参阅附录 E 和附录 F。

说明 为简明起见（同时也是为了让有关的路径名更简短一些），本章在讨论与 Windows 系统有关的问题时将假设 MySQL 软件是安装在 C:\mysql 子目录里的。但 MySQL 软件的 Windows 安装向导通常会把 MySQL 发行版本安装到 C:\Program Files\MySQL\MySQL Server X.Y 子目录下，其中的 X.Y 是诸如 5.0 或 5.1 之类的版本系列号。如果使用了那样的安装位置，记得要对本章各有关示例当中的 Windows 路径名做必要的调整。

12.1 安装 MySQL 软件后的初始安防设置

我们的讨论将从需要在安装 MySQL 软件后立刻进行的一项系统管理任务开始，确保服务器只能由授权用户访问。为此，必须知道有哪些 MySQL 用户账户会在安装过程中被创建，然后给那些需要

保留的用户账户加上口令，删除其余的用户账户。

MySQL 软件的安装过程将为 MySQL 服务器创建一个数据目录并在其中生成两个数据库。

- ❑ 一个mysql数据库，里面容纳着用来控制客户对服务器的访问权限的各种权限数据表。
- ❑ 一个test数据库，这是一个用于测试目的的数据库。

如果这是你在某个主机上第一次安装 MySQL（具体步骤见附录 A），在 mysql 数据库中的权限数据表里将会出现几个处于其初始状态的账户，这些账户允许任何人在不使用口令的情况下连接服务器。这当然不安全，所以应该尽快给这些账户加上口令。如果是在一个现有 MySQL 系统的基础上安装一个新版本对之进行升级，权限数据表应该是已经设置过的，口令应该都设置妥当了。如果是在一台已经安装过 MySQL 的主机上再次把 MySQL 安装到一个新位置，将需要为新服务器设置口令。但要注意可能遇到新服务器从为老服务器创建的选项文件里提取口令占为己有的麻烦，详见 12.1.3 节中的讨论。

为便于展开讨论，在后续讨论内容所涉及的示例操作中，将假设你的 MySQL 服务器正运行在一台主机名是 cobra.snake.net 的机器上，而你将从同一台机器去连接服务器。当你在操作步骤中看到这个主机名的时候，请替换为你自己的服务器主机名。示例中还将假设你的 MySQL 服务器已经启动运行，只要连接它就可以开始进行系统管理操作。

说明 有些 MySQL 安装向导会在安装过程中向你提供一个创建口令的选项，但即便你使用的是那样的安装向导，本章内容也可以帮助你加深对初始 MySQL 用户账户的理解。接下来的讨论将假设你还没有设置过任何口令。还有一些安装向导可能会提供一个只创建一部分这里所讨论的初始账户的选项，如果真是那样，你就用不着和那些没创建出来的初始账户打交道了。

12.1.1 为初始MySQL账户设置口令

本节将介绍如何检查权限数据表里都有哪些账户以及如何设置它们的口令。

MySQL 软件的安装过程将在 mysql 数据库中的权限数据表里创建两种初始账户。

- ❑ 以root为用户名的账户。这些是用于系统管理的超级用户账户。root账户拥有全部的权限，可以用来做任何事情，其中包括删除所有的数据库和关闭服务器。（MySQL超级账户和Unix超级账户的名字都是root这一事实纯属巧合。虽然这两种超级账户都拥有最高的权限，但彼此毫无瓜葛。）
- ❑ 用户名是空白的账户。这些是“匿名”账户，任何人都可以使用匿名账户连接服务器而无需拥有账户。匿名账户的权限往往很有限，只能用来完成一小部分操作。（在Windows系统上，MySQL 5.0.36/5.1.16之前的版本可能还包括一个拥有超级用户权限的匿名账户，应该按照本节后面指示对它进行处置。）

MySQL 服务器上的已知用户全都列在其 mysql 数据库中的 user 数据表里，由安装过程创建的初始用户也不例外。在默认的情况下，这些账户都没有口令，因为 MySQL 认为应由系统管理员来设置。因此，安装 MySQL 软件后的首要任务之一就是设置口令。否则，非授权用户就可以通过利用 root 账户去连接服务器的办法获得超级用户权限。在加强了这些初始账户的安全措施之后，应该再另行创建一些账户提供给你的用户团体成员，让他们使用你设置的用户名和你认为他们应该具备的访问权限去连接服务器。我们将在 12.4 节介绍如何创建新账户和修改现有账户。

user 数据表里的每一项（数据行）包含着一个 Host 值，它限定了用户可以从哪台主机来连接服务器；还包含着一个 User 值和一个 Password 值，它们是用户从指定主机连接服务器时必须给出的用户名和口令。user 数据表还有其他一些数据列，用来表明各个账户有哪些超级权限。

为了查看都有哪些初始账户以及它们是否有口令，请使用 root 账户连接服务器并查询 mysql.user 数据表。因为新安装的 root 账户没有初始口令，所以用不着给口令也应该连接得上：

```
% mysql -u root
mysql> SELECT Host, User, Password FROM mysql.user;
+-----------------+------+----------+
| Host            | User | Password |
+-----------------+------+----------+
| localhost       | root |          |
| cobra.snake.net | root |          |
| 127.0.0.1       | root |          |
| localhost       |      |          |
| cobra.snake.net |      |          |
+-----------------+------+----------+
```

你在自己的服务器上看到的输出结果是否和如上所示的完全一样并不重要，重要的应该为看到的每一个有着空白 Password 值的账户设置一个口令。

在 Unix 系统上，数据目录的初始化工作是在安装过程中由 mysql_install_db 脚本完成的。如果在 Linux 系统上使用一个 PRM 软件包或者是在 Mac OS X 系统上使用一个 DMG 软件包来安装 MySQL，mysql_install_db 脚本将自动运行。否则，就应该由你本人来运行它。这方面的细节请参阅附录 A。

mysql_install_db 脚本的任务之一是在 mysql 数据库里创建必要的权限数据表。在一台名是 cobra.snake.net 的服务器主机上，mysql_install_db 脚本对 user 数据表进行的初始化将在这个数据表里创建出以下账户。

主　　机	用　　户	口　　令	超级用户权限
localhost	root		全部
127.0.0.1	root		全部
cobra.snake.net	root		全部
localhost			无
cobra.snake.net			无

user 数据表里的这些初始账户分别允许客户程序按以下方式连接服务器。

- 那几个root项允许使用主机名localhost、127.0.0.1或cobra.snake.net去连接本地MySQL服务器。比如说，下面两条命令当中的任何一个都可以让你在登录到cobra.snake.net主机后使用mysql程序以root用户的身份连接上服务器：

```
% mysql -h localhost -u root
% mysql -h cobra.snake.net -u root
```

  作为 root 用户，你将拥有全部的权限，可以执行任何操作。

- 有着空白User值的那几项都是匿名账户。它们允许使用主机名localhost或cobra.snake.net在无需给出任何用户名的情况下连接本地MySQL服务器：

```
% mysql -h localhost
% mysql -h cobra.snake.net
```

匿名用户不具备任何超级用户权限。

在 Windows 系统上，数据目录和 mysql 数据库已包括在 MySQL 发行版本当中并接受过预初始化处理，其初始账户的情况与 Unix 系统上的可能不太一样。Windows 系统上的 user 数据表项如下表所示。

主　机	用　户	口　令	超级用户权限
localhost	root		所有
127.0.0.1	root		所有
localhost			依版本而定

user 表里的这些账户记录可以让客户程序建立以下连接。

- ❑ 从本地主机以root身份连接MySQL服务器。作为root用户，你将拥有全部的权限，能执行任何一种操作。
- ❑ 不必给出任何用户名就能从本地主机以匿名方式连接服务器。在MySQL的新版本里，这个账户没有任何超级用户权限。但在MySQL 5.0.36/5.1.16版之前，这个账户拥有与root用户完全一样的超级用户权限，可以执行任何一种操作。对于这种情况，你不仅应该给这个账户加上一个口令，还应该撤销那些权限，或者干脆把整个账户彻底删掉。

另一个权限表（db 数据表，这里没有给出）包含的权限信息允许匿名用户使用 test 数据库和任何名字以 test-开头的其他数据库。

本节剩下的内容将介绍如何为 root 用户和匿名用户设置口令。下面的例子使用的账户只是用于演示，你可根据系统的实际账号在相应的 SQL 语句里对它们替换。

根据你分配的口令，你可能还需要让服务器重新加载授权表才能让你的修改生效。服务器是使用内存里的权限表副本来进行访问控制的。如果你直接去修改 user 权限表中的口令，服务器可能不知道你修改了些什么，所以你必须明确地告诉它去重新读取权限表。

设置口令的第一种办法是以 root 用户的身份连接服务器，查出哪些账户没有口令，然后使用 SET PASSWORD 语句为它们分别设置一个口令。我们不妨假设你已连接到服务器并发现以下账户还没有口令：

```
% mysql -u root
mysql> SELECT Host, User FROM mysql.user WHERE Password = '';
+-----------------+------+
| Host            | User |
+-----------------+------+
| localhost       | root |
| cobra.snake.net | root |
| 127.0.0.1       | root |
| localhost       |      |
| cobra.snake.net |      |
+-----------------+------+
```

这些账户的口令都可以用 SET PASSWORD 语句来设置。每一条 SET PASSWORD 语句里，账户的名字必须以'user_name'@'host_name'的格式给出，其中的 user_name 和 host_name 分别是 user 权限表里的 User 和 Host 数据列的值（如果 User 值是空白，user_name 值将是空字符串）。下面这些语句将为刚才的 root 和匿名用户设置口令：

```
mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD('rootpass');
mysql> SET PASSWORD FOR 'root'@'cobra.snake.net' = PASSWORD('rootpass');
mysql> SET PASSWORD FOR 'root'@'127.0.0.1' = PASSWORD('rootpass');
mysql> SET PASSWORD FOR ''@'localhost' = PASSWORD('anonpass');
mysql> SET PASSWORD FOR ''@'cobra.snake.net' = PASSWORD('anonpass');
```

设置口令的第二种办法是用 UPDATE 语句直接修改 user 权限表。这个办法可以用来为某给定 User 值的所有账户（不管它们的 Host 值是什么）设置一个口令，因为可以一次修改多个账户。下面这些语句将为所有的 root 账户和所有的匿名用户账户设置口令：

```
mysql> UPDATE mysql.user SET Password=PASSWORD('rootpass') WHERE User='root';
mysql> UPDATE mysql.user SET Password=PASSWORD('anonpass') WHERE User='';
mysql> FLUSH PRIVILEGES;
```

如果使用 SET PASSWORD 语句来改变口令，服务器将觉察到你对权限表进行了修改，它将自动重新读入权限表的新内容去更新内存里的副本。如果使用 UPDATE 语句直接修改 user 权限表，就必须明确地通知 MySQL 服务器重新加载这个权限表。上面例子里紧跟在 UPDATE 语句后面的那条 FLUSH PRIVILEGES 语句就是用来更新权限表的。

在 Windows 系统上，如果某个匿名用户账户具备和 root 一样的超级用户权限，它的权限将大于你希望它具备的，会成为一个严重的安全隐患。如果你想保留那个账户但剥夺它的超级用户权限，你可以撤销它们。下面这条语句可以用来查出账户有哪些权限：

```
mysql> SHOW GRANTS for ' '@'localhost'
```

如果账户没有任何超级用户权限，上面这条语句的输出将为如下所示，而你无需采取任何行动：

```
+-------------------------------------+
| Grants for @localhost               |
+-------------------------------------+
| GRANT USAGE ON *.* TO ''@'localhost' |
+-------------------------------------+
```

如果账户有超级用户权限，你将看到如下所示的输出：

```
+-------------------------------------------------------------+
| Grants for @localhost                                       |
+-------------------------------------------------------------+
| GRANT ALL PRIVILEGES ON *.* TO ''@'localhost' WITH GRANT OPTION |
+-------------------------------------------------------------+
```

要想撤销那个账户的权限，请使用这些 REVOKE 语句：

```
mysql> REVOKE ALL ON *.* FROM ''@'localhost';
mysql> REVOKE GRANT OPTION ON *.* FROM ''@'localhost';
```

处理匿名用户账户的另一个办法是把它们彻底删掉，如果你根本用不上匿名账户，我强烈建议你这么做。删除用户账户的工作可以用 DROP USER 语句完成：

```
mysql> DROP USER ''@'localhost';
mysql> DROP USER ''@'cobra.snake.net';
```

删除匿名用户账户的最大好处是它将大大简化非匿名账户的设置工作。否则，将会出现奇怪的现象，13.2.3 节对这一现象背后的细节进行了描述。

在执行完 REVOKE 和 DROP USER 语句之后，MySQL 服务器将自动地重新读入权限表，你不必明

确地发出任何 FLUSH PRIVILEGS 语句。

给账户设置好口令（并视情况重新加载了权限表）之后，连接服务器时必须给出正确的口令。值得一提的是，不知道口令的人将无法以 root 用户的身份连接服务器：

```
% mysql -u root
ERROR 1045 (28000): Access denied for user 'root'@'localhost'
(using password: NO)
% mysql -p -u root
Enter password: rootpass
mysql>
```

必须给出正确的口令才能成功连接服务器，不仅对 mysql 程序来说是如此，对 mysqladmin 和 mysqldump 等其他程序来说也是如此。为简明起见，本章后面内容里的很多例子都省略了-u 和-p 选项，但你在连接服务器时千万不要忘记把你的口令加进去。

12.1.2 为第二个服务器设置口令

前面的描述的前提是，你是在以前没有安装 MySQL 的系统上建立口令。然而，如果 MySQL 已安装在某个位置，若为安装在同一计算机另一位置的新服务器设置口令，你会发现，当你试图在没有口令的情况下连接新的服务器时，它会拒绝你的企图，并给出下列错误：

```
% mysql -u root
ERROR 1045 (28000): Access denied for user 'root'@'localhost'
(using password: YES)
```

奇怪！当你没有指定口令时，为什么服务器说它收到了一个口令？这通常是指你设置了一个选项文件，列出了访问以前所安装的服务器的口令。mysql 会找到选项文件并自动使用这里列出的口令。为了重写并明确指定“没有口令”，使用-p 选项，当 mysql 提示输入口令时按下 Enter 键：

```
% mysql -p -u root
Enter password:              ← just press Enter
```

可以在调用 mysqladmin 以及其他 MySQL 客户程序时采用这种决策。

12.10 节讨论了多个服务器的使用。

12.2 安排 MySQL 服务器的启动和关停

MySQL 管理员的一个基本目标是保证服务器 mysql 尽可能地运行，以便你的用户能够访问。然而有时候必须关停服务器。例如，当你重定位数据库，不想要服务器同时修改数据库中的数据表时，就需要关闭服务器。若想保持服务器运行，但有时又必须关停它，这是本节不能解决的问题。但是至少我们能讨论如何让服务器启动和停止，使你能执行认为任何操作。这些过程的许多方面对 Unix 和 Windows 是不一样的，所以下面的讨论是分开进行的。

12.2.1 在Unix上运行MySQL服务器

在 Unix 上，MySQL 服务器可以从命令手动启动。还可以安排服务器在系统启动时作为标准启动过程的一部分自动运行。事实上，在正常工作条件下，当一切按你想要的方式设置好之后，你可按这种方法启动服务器。但是在讨论如何启动服务器之前，让我们先考虑一下哪个登录账号应该在服务器启动时运行。在 Unix 这样的多用户操作系统上，你可以选择使用哪个登录账号运行服务器。例如，

如果手动启动服务器，则作为 Unix 用户（你登录时的身份）运行该服务器。例如，如果以 paul 登录并启动服务器，则作为 paul 运行；如果使用 su 命令，使用户转换至 root 后启动服务器，则作为 root 运行。

你应该记住 MySQL 服务器在 Unix 下启动过程的两个目标。

以 root 以外的其他用户运行服务器。以给定用户运行服务器，意味着服务器进程和用户的 Unix 登录账号的 ID 有关，它具有读和写文件系统中文件的用户权限。这表明有一定的安全性，特别是对于作为 root 用户运行的进程，因为 root 可以做任何事情，所以有危险性。避免这些危险的一种方法是使服务器撤回其专门的权限。作为 root 启动的进程能将它们的用户 ID 改成另外的账号，这样就放弃了 root 的权限，成了普通的没有权限的用户，也就使该进程减少了危险。总之，应该限制任一进程的权力，除非它真正需要 root 访问权，而且 mysqld 不需要。服务器必须访问和管理 MySQL 数据目录的内容，然而几乎很少。这意味着，如果作为 root 启动服务器，应该让它改变用户，在启动过程中作为无权限用户运行。

应该固定使用同一个用户账户来运行 MySQL 服务器。让 MySQL 服务器每次以不同用户的权限运行的做法会让整个数据库系统变得很不稳定。这将导致创建在数据目录里的文件和子目录有不同的属主，而这意味着无论 MySQL 服务器以哪个用户的权限运行，总有一部分数据库或数据表是它无法访问的。只要固定使用同一个用户账户来运行 MySQL 服务器，就可以从根本上避免这类问题。

1. 使用一个低权限登录账户来运行MySQL服务器

专门创建一个有别于 root 账户的低权限账户来从事与 MySQL 有关的活动有以下一些好处。

- 只要不使用root账户来运行MySQL服务器，就算黑客利用它的安全漏洞攻陷了它也不会获得root账户的权限。
- MySQL服务器在运行期间创建的文件将属于mysql账户而不属于root账户。这样一来，拥有FILE权限的MySQL用户就无法通过MySQL服务器对root账户拥有的文件执行写操作。系统里的这类文件数量越少越好。
- 以低权限用户登录去执行MySQL系统管理任务要比以root用户登录更加安全。如果以root登录用户，万一在执行与文件系统有关的操作时犯了错误，就有可能导致灾难性后果。
- 另行创建一个专门用来从事与MySQL有关的活动的账户在概念上和操作上都更加清晰。系统里的哪些东西与MySQL有关将一目了然。比如说，在集中存放crontab文件的子目录里，用来运行MySQL服务器的mysql用户账户将有一个它自己的文件。否则，与MySQL有关的cron作业将被列在root账户的crontab文件里，与root账户定期执行的各种任务混杂在一起。

为了设置作为无权限的非 root 用户运行服务器，要遵守下列步骤。

(1) 关停正在运行的服务器：

```
% mysqladmin -p -u root shutdown
```

(2) 选择用于运行 mysqld 的登录账号。可以指定一个专用于 MySQL 的组名。我们把这些用户和组名都称为 mysql。如果你使用不同的名字，可在本书中见到以 my 作为用户或组名的任何地方替代它们。例如，如果你在自己的账号下安装了 MySQL（因为在你系统上没有专门管理的权限），将有可能在你自己的用户 ID 下运行服务器。在这种情况下，用你自己的注册名和组名替换 mysql。

(3) 如有必要，使用系统通常建立账号的步骤建立你所选用户名的登录账号。你必须以 root 用户身份来做这些工作。

如果决定使用一个名为 mysql 的账户来运行服务器，说不定已经有人替你把它创建出来了。如果在 Linux 系统上使用一个 RPM 软件包来安装 MySQL，安装过程会自动创建这个账户。Mac OS X 的最近几个版本也准备了一个已经创建好的 mysql 账户。其他系统有可能也会这么做。

(4) 修改 MySQL 数据目录、子目录和该目录下文件的用户和组的所有权，以便 mysql 用户占用。例如，如果数据目录是/usr /local /mysql /data，可以按照下示为那个目录及其内容设置所有权（必须作为 root 运用下示命令）：

```
# chown -R mysql /usr/local/mysql/data
# chgrp -R mysql /usr/local/mysql/data
```

(5) 良好的安全预防措施是设置数据目录的访问模式，防止其他外人侵入。为了做这些工作，修改其许可权限，使得只有 mysql 可以使用它。如果数据目录是/usr /local /mysql /data，可以断开所有“组”和“其他”的许可权限，设置该目录内和目录下的每一事情只有 mysql 可访问，如下所示：

```
# chmod -R go-rwx /usr/local/mysql/data
```

最后两步实际上是更全面锁定步骤的一部分，详情请见 13.1.2 节。在制定所有权和模式分配时，特别是 MySQL 具有非标准组织时，一定要检查那一节中的附加说明。

完成前面的步骤后，一定要用--user=mysqladm 选项启动该服务器，所以由 root 调用时，将把用户 ID 转换成 mysql。（对于作为 root 手动运行服务器，以及在系统启动过程期间设置要调用的服务器时，都是如此。Unix 系统以 Unix root 用户执行启动操作，所以按这一部分步骤启动的任何进程默认地用 root 权限执行。）保证始终如一地指定用户的最佳方法是在选项文件（服务器会读取）中把它列出。例如，把下列各行放在/etc/my.cnf 中：

```
[mysqld]
user=mysql
```

有关选项文件的详细内容参见 12.2.3 节。

如果以 mysql 登录，然后启动服务器，那么在选项文件中出现 user 行时，会产生一个警告，表明只有 user 用户才能使用选项。这意味着服务器没有能力改变其用户 ID，而将以 mysql 身份运行。这正是你希望的事情，不用理会该警告。

2. 在Unix上启动服务器

当你确定以什么登录账号运行服务器后，对如何启动服务器可有几种选择。可以从命令行手动运行服务器，也可在系统启动过程中自动运行。做这个工作的方法包括下述几种。

- 直接调用mysqld。这大概是最不常见的方法，不予讨论，只是说明**%mysqld--verbose--help**是一个寻找服务器支持什么启动选项的有用命令。
- 调用mysqld_safe脚本程序。mysqld_safe调用服务器，然后监视服务器，当其意外中断时重新启动它。mysqld_safe一般用于BSD格式的Unix版本上，还可由非BSD系统和Mac OS X系统上的mysql.server使用。

  mysqld_safe 使错误信息和其他诊断输出从服务器改至 Syslog 或数据目录下的一个文件中，以便产生一个错误日志。如果你向文件发送错误输出，mysqld_safe 会设置错误日志的所有权，使其由--user 选项命名的用户占用。当你在不同的时间使用不同的--user 值时可能会导致错误，mysqld_safe 写至错误日志时会由于“许可权被拒绝”而失败。这可能特别成问题，因为你检查错误日志来查看故障时，不会包含关于原因的有用信息。如果发生该问题，去除错误日志，重新调用 mysqld_safe。

- 调用mysql.server脚本程序。mysql.server通过执行mysqld_safe启动服务器。该脚本程序可以用变量start或stop来调用，指示你是要服务器启动还是关停。它用作mysqld_safe的包装器，常用于在使用System V方法的系统上，System V方法把启动和关停脚本程序编排入几个目录中。每个目录对应于特定的运行级，并包括机器进入或退出那个运行级时要调用的脚本程序。
- **如果需要协调启动多个 MySQL 服务器，请使用 mysqld_multi 脚本。**这个脚本将读取一个选项文件，其内容是你为多个服务器列出的启动参数。还可以通过这个脚本单独启动或关闭它们当中的某一个服务器，或者检查它是否正在运行。这个启动脚本比其他脚本都复杂得多，12.10 节再对它进行详细讲解。

mysqld_safe 和 mysqld_multi 脚本程序安装在 MySQL 安置目录下的 bin 目录中，也可以在 MySQL 源发行版本的 scripts 目录中找到。mysql.server 脚本程序安装在 MySQL 安置目录下的 share/mysql 目录中，也可以在 MySQL 源发行版本的 support_files 目录中找到。如果要使用 mysql.server，需要把它复制至合适的运行级别目录中，然后让它可以执行。（有些安装方法会替你安装好 mysql.server 脚本。Linux RPM 和 Mac OS X DMG 软件包就是这么做的。）如果 MySQL RPM 软件包来自另一个供应商，可能会有一个类似的启动脚本安装在一个不同的文件名下，比如 mysqld。

为了让启动脚本在系统开机时执行而需要做出的具体安排取决于使用的系统平台。请仔细阅读下面给出的操作示例，并根据自己的系统启动过程来选择与之最相符的操作步骤。

对于 BSD 格式的系统，在/etc 目录中通常有几个文件用于启动服务器。这些文件的名字通常以 rc 开头，如 rc.local 名字的文件（或者是一些类似的名字），专门用于启动本地安装的服务器。在这样的基于 rc 的系统上，可以把如下所示的各行增加至 rc.local 中，以便启动服务器：

```
if [ -x /usr/local/bin/mysqld_safe ]; then
  /usr/local/bin/mysqld_safe &
fi
```

如果你的系统上 mysqld_safe 的与以往不同，则对各行作相应的修改。

对于 System V 格式的系统，可以安装 mysql.server，并把它复制至相应 /etc 下的运行级别目录中。如果你运行 Linux 并由 RPM 文件安装了 MySQL，这很有可能已经完成了。否则，就把脚本程序安装在具有你想用的名称的主启动脚本目录中，保证该脚本程序可以执行，并且把其链接放在相应运行级的目录中。

说明　有相当一部分系统管理员会把安装到某个运行级子目录里的 mysql.server 脚本重新命名为 mysql，但我仍将把它称为“mysql.server 脚本”，这是为了让你们能够明白我指的到底是什么。

运行级别的目录布置因系统而变，所以必须查看你的系统如何组织它们。例如，Solaris 下，一般的多用户运行级是 2，主脚本目录是/etc/init.d，而运行级目录是/etc/rc2.d，所以各命令如下所示：

```
# cp mysql.server /etc/init.d/mysql
# cd /etc/init.d
# chmod +x mysql
# cd /etc/rc2.d
# ln -s ../init.d/mysql S99mysql
```

在系统启动时，启动过程自动调用具有 start 参数的 S99 mysql 脚本。

许多 Linux 变体有类似的目录集，但是它们编排在/etc/init.d 和/etc/rc.d 下面。Linux 系统一般具有用于启动脚本管理用的 chkconfig 命令。使用它有助于安装 mysql.server 脚本，而不必手动运行如刚才所示的命令。下面的说明示出了如何使用 mysql 名称把 mysql.server 安装到启动目录中。

(1) 把 mysql.server 脚本从其现有位置复制至 init.d 目录中，并使其可执行：

```
# cp mysql.server /etc/init.d/mysql
# chmod +x /etc/init.d/mysql
```

(2) 寄存该脚本并启动它：

```
# chkconfig --add mysql
# chkconfig mysql on
```

(3) 为了检查该脚本是否正确启用，运行具有--list 选项的 chkconfig：

```
# chkconfig --list mysql
mysql           0:off   1:off   2:on    3:on    4:on    5:on    6:off
```

其输出指示该脚本程序以运行级 3、4 和 5 自动执行。

如果没有 chkconfig，可以使用和 Solaris 所用类似的程序，但其路径名稍有不同。以运行级 3 启动脚本程序时，使用下列命令：

```
# cp mysql.server /etc/init.d/mysql
# cd /etc/init.d
# chmod +x mysql
# cd /etc/rc.d/rc3.d
# ln -s /etc/init.d/mysql S99mysql
```

在 Mac OS X 下面，启动过程有所不同。/Library/StartupItems 和/System/Library /StartupItems 目录包括在系统启动时启动服务器用的子目录。在 MySQL 官方网站为 Mac OS X 系统提供的 DMG 软件包里有一个安装向导，它会把对应于 MySQL 服务器的开机启动项添加到这些子目录当中的某一个里去。

12.2.2　在Windows上运行MySQL服务器

在 Windows 系统上，既可以从命令行以手动方式启动 MySQL 服务器，也可以把 MySQL 服务器安装为一项 Windows 服务。如果把它安装为一项 Windows 服务，可以设置该项服务在 Windows 启动时自动运行并从命令行去控制它，也可以使用 Windows 服务管理器（Windows Service Manager）去控制它。

供 Windows 系统安装使用的 MySQL 发行版本收录了好几种 MySQL 服务器程序，它们是用不同的选项编译。可以在附录 A 里找到一份关于它们的汇总。在接下来的讨论内容里，举例时将使用“mysqld”作为 MySQL 服务器程序的名字，但你所使用的 MySQL 发行版本有可能还提供了另外几种名字各不相同的 MySQL 服务器，例如 mysqld-nt 或 mysql-debug。

Windows 系统上 MySQL 服务器提供了两种在 Unix 系统上没有的连接方式。

- 通过命名管道建立的连接，如果服务器在启动时使用了--enable-named-pipe选项的话。
- 通过命名管道建立的连接，如果服务器在启动时使用了--shared-memory选项的话。

在 MySQL 5.1.21 之前的版本里，只有名为 mysqld-nt 和 mysql-debug 的服务器支持命名管道。（“nt”源自 Windows 2000 的前身 Windows NT 操作系统，而命名管道是从 Windows NT 开始新增的一项功能。mysql-debug 类似于 mysql-nt，但增加了调试支持。）从 MySQL 5.1.21 版开始，所有基于 Windows 平台的 MySQL 服务器都已经内建了命名管道支持，它们的名字也不再加上“-nt”字样以示

区别了。

1. 在Windows上手动运行服务器

为了手动启动服务器，在控制台窗口由命令行调用它：

```
C:\> mysqld
```

如果想要错误信息进入控制台窗口，而不是进入错误记录（在数据目录中的 host_name.err 文件），使用--console 选项：

```
C:\> mysqld --console
```

当从命令行运行 MySQL 服务器时，在服务器退出执行之前可能不会再看到另一个命令提示符，这是正常的。它只意味着需要打开另一个控制台窗口去运行 MySQL 客户程序。

如果在启动命令里加上了--shared-memory 选项,服务器将允许本地客户使用共享内存来建立连接。类似地，如果服务器具备命名管道支持，用--enable-named-pipe 选项将允许本地客户通过命名管道来建立连接。

如果想停止服务器的运行，可以使用 mysqladmin 工具程序：

```
C:\> mysqladmin -p -u root shutdown
```

2. 把MySQL服务器运行为一项Windows服务

在 Windows 系统上，MySQL 服务器可以用下面的命令安装完一项 Windows 服务：

```
C:\> C:\mysql\bin\mysqld --install
```

这条命令使用了 MySQL 服务器程序的完整路径名。如果把 MySQL 服务器安装在了另外一个地方，请对这条命令里的路径名做相应的修改。

服务安装命令本身不会立刻启动 mysqld 程序运行，但它将导致 mysqld 在 Windows 启动时自动运行。如果你更喜欢自己决定应该在何时启动该项服务而不希望它在系统开机时自动运行，把 MySQL 服务器安装为一项“人工”服务即可：

```
C:\> C:\mysql\bin\mysqld --install-manual
```

作为一个基本原则，当把一个服务器安装为一项 Windows 服务的时候，应该把所有必要的选项列在一个选项文件里而不是在命令行上给出（参见 12.2.3 节）。不过，以参数的形式指定服务名和选项文件还是允许的，详见接下来的讨论。这在安装了多个 MySQL 服务器作为 Windows 服务的时候将非常有用（参见 12.10 节）。

当把一个 MySQL 服务器安装为一项 Windows 服务的时候，默认的服务名是 MySQL。（服务名不区分大小写。）可以在--install 选项的后面明确地指定一个服务名：

```
C:\> C:\mysql\bin\mysqld --install service_name
```

每一项 Windows 服务都必须有一个独一无二的名字，不使用默认的 MySQL 的好处之一是可以把多个 MySQL 服务器运行为 Windows 服务。服务名会对服务器在启动时将从选项文件读取哪些选项组产生影响。

- 如果既没有给出任何service_name参数，也没有使用MySQL作为服务名，服务器将使用默认的服务名（MySQL）并从标准选项文件读取[mysqld]选项组。
- 如果利用service_name参数指定了一个不是MySQL的服务名，服务器将使用那个名字作为服务名并从标准选项文件读取[mysqld]和[service_name]选项组。

在安装服务器时，如果指定了一个服务名，还可以指定一个--default-file 选项作为命令行上的最后一个选项：

```
C:\> C:\mysql\bin\mysqld --install service_name  --default-file=file_name
```

这意味着又多了一种向服务器提供各种选项的手段。服务器在启动时将使用这个文件名去寻找选项文件，并且只从那个文件的[mysqld]选项组读取选项。这个语法要求必须提供一个服务名，哪怕使用的是默认的服务名，也必须把 service_name 参数的值设置为 MySQL。

在服务名的后面只允许给出一个非--default-file 选项，但--default-file 选项显然更灵活，因为可以把需要用到的选项都放在由该选项指定的文件里。

服务器以一个服务安装后，可以使用服务名控制它。这可以由命令行完成，如果你喜欢图形界面，也可由 Windows 服务管理程序完成。服务管理程序可以在 Windows 控制面板内从服务项找到，也可在控制面板的管理工具栏中找到，这取决于 Windows 的版本。

为了从命令行启动或停止服务，使用下列命令：

```
C:\> net start MySQL
C:\> net stop MySQL
```

如果你使用服务管理程序，它会提供一个窗口显示出你要知道的服务列表，以及其他信息，例如每个服务是否在运行，是自动还是手动的。为了启动或停止 MySQL 服务器，选择服务表中它的登记项，然后选择合适的按钮或菜单项。

也可以从命令行用 mysqladmin shutdown 关停服务器。

说明 虽然能使用服务管理程序或命令提示符处的命令控制服务，但应该力求避免两种方法之间的交互。要保证当你从提示符处调用服务相关命令时关闭服务管理程序。

为了从服务表中删除 MySQL，先关停正在运行的服务器，然后发出下列命令：

```
C:\> mysqld --remove
```

这条命令将删除有默认服务名 MySQL 的那项 MySQL 服务。如果想明确地表明应该删除哪一项服务，在--remove 选项的后面给出它的名字即可：

```
C:\> mysqld --remove service_name
```

12.2.3 指定服务器启动选项

在任一平台上，当你调用服务器时有两个主要方法可指定启动选项。

- 可以在命令行上列出选项。在这种情况下，既能使用长格式，也能使用短格式的任一选项，这两种格式都可使用。例如，可以使用--user=mysql或-u mysql。
- 可以在选项文件中列出选项。每行指定一个选项，只能使用长格式选项，并且没有领头的虚线：

```
[mysqld]
user=mysql
```

F.2.2 节对选项文件的格式、语法以及服务器会到哪些地方去寻找它们进行了比较全面的讨论。

两个选项规范方法不是互斥的。服务器在选项文件和命令行上寻找选项，命令行上的选项优先。

这通常是使用选项文件的最简单方法，可用于任何启动方法。一旦选项文件中放置了选项，则每次启动服务器时都起作用。在命令行上列出选项，仅在手动启动服务器时或者用 mysqld-safe 启动时起作用。它对 mysql.server 不起作用，因为它只支持命令行上的 start 和 stop 选项。此外，存在几个例外，如果使用--install 或--install-manual 以一个服务安装一个 Windows 服务器，则不能在命令行上指定启动选项。（这些例外将在 12.2.2 节的第 2 小节中讨论）。

服务器会到哪些地方去寻找选项文件取决于 MySQL 版本（参见 F.1.3 节）。但 Unix 系统上的/etc/my.cnf 文件和 Windows 系统上的 C:\my.ini 文件适用于 MySQL 5.0 和它以后的所有版本。如果打算使用的文件尚不存在，请把它创建为一个普通文本文件。

总之，服务器启动选项放在[mysqld]选项组内。例如，为了指出你要服务器以 mysql 运行而使用/usr /local /mysql 数据库目录的位置，可以把下列组放入选项文件中：

```
[mysqld]
user=mysql
basedir=/usr/local/mysql
```

这相当于用命令行的选项启动服务器：

```
% mysqld --user=mysql --basedir=/usr/local/mysql
```

表 12-1 列出了 MySQL 服务器以及各种 MySQL 服务器启动程序所使用的标准选项组。mysqld 那一行也适用于有着变体名字的服务器，如 Windows 系统上的 mysqld-debug。

服务器所用的选项组列表及其服务器启动程序如表 12-1 所示。

表12-1　服务器程序使用的选项组

程　序	程序所用的选项组
mysqld	[mysqld], [server], [mysqld-X.Y]
mysqld_safe	[mysqld], [server], [mysqld_safe], [safe_mysqld]
mysql.server	[mysqld], [server], [mysql_server], [mysql.server]
libmysqld	[server], [embedded], [*appname*_server]

mysqld 那一行里的[mysqld-X.Y]表示服务器将读取为指定版本系列号准备的选项组。MySQL 5.0 系列服务器将读取[mysqld-5.0]选项组，MySQL 5.1 系列服务器将读取[mysqld-5.1]选项组，依次类推。

mysql_safe 脚本读取[safe_mysql]选项组是为了保持兼容性。现如今的 mysql_safe 脚本在 MySQL 4.0 以前的版本里叫做 safe_mysql。

mysql.server 脚本只寻找并读取各有关选项文件里的 basedir、datadir 和 pid-file 选项值。

libmysqld 那一行对应着嵌入式 MySQL 服务器库，这个库可以链接到其他程序当中以生成基于 MySQL 的应用程序，这种应用程序在运行时不需要一个独立型 MySQL 服务器。（第 7 章讨论了如何编写使用了嵌入式服务器的应用程序。）[appname_server]代表着一个只能由内含嵌入式服务器的 appname 应用程序读取的选项组。（这只是一个建议。从这个选项组读取选项的具体操作必须由应用程序本身实现。）

在 Windows 系统上，如果把一个 MySQL 服务器安装为一项 Windows 服务但没有使用默认的服务名，该服务器读取选项组的行为将受到影响。详见 12.2.2 节的第 2 小节。

在把选项放到某个选项组里去时，应该选择在上下文或将使用的上下文中肯定会读取的选项组。适用于所有服务器（无论它们是独立型的还是嵌入式的）的选项应该放到[server]选项组里。只适用

于独立型或是嵌入式服务器的选项应该分别放到[mysqld]或[embedded]选项组里。类似地，只适用于启动脚本的选项应该分别放到[mysql_safe]或[mysql.server]选项组里去。

如果打算使用 mysql_safe 或 mysql.server 脚本启动服务器，还可以通过编辑脚本把选项直接传递给服务器。但应该把这作为最后的手段，因为它有一个明显的弊病：在每次升级 MySQL 的时候，修改过的脚本都会被新版本覆盖掉，而你将不得不重新再改一遍。

12.2.4 关闭服务器

手动关闭服务器时使用 mysqladmin：

```
% mysqladmin -p -u root shutdown
```

该工作对于 Unix 和 Windows 都适用。如果你以 Windows 下的一个服务安装服务器，也可以从命令行手动行止服务器：

```
C:\> net stop MySQL
```

或者使用服务器管理程序提供的图形界面选择和停止服务器。

如果已经把 MySQL 服务器设置成在系统开机时自动启动，应该不需要在系统关机时做任何事情去停止它。在系统关机时，BSD Unix 系统会发出一个 TREM 信号正常结束运行，那些进程应该对此信号做出正确的响应（如果它们不能正常地结束运行，就会被无情地“杀掉”）。mysqld 进程在收到这个信号时做出的响应是结束运行。

对于 System V 格式的 Unix 系统，它用 mysql.server 启动服务器，关闭过程将调用具有 stop 参数的脚本，通知服务器关闭。还可以调用自己的脚本手动关闭服务器。举例来说，如果以 /etc/rc.d/init.d/mysql 安装了 mysql.server，可以按下面的方法调用它（必须作为 root 进行这项工作）：

```
# /etc/init.d/mysql stop
```

如果以 Windows 服务运行 MySQL 服务器，服务器管理程序会自动告知服务器在系统关停时停止。如果你没有以一个服务运行该服务器，应该在关闭 Windows 之前在命令行上用 mysqladmin shutdown 或 net stop MySQL 手动停止服务器。

12.2.5 当你未能连接至服务器时重新获得服务器的控制

在某些环境下，当发现连接不上服务器时，可能需要手动重启。这就形成了一种困局：要想关闭 MySQL 服务器，必须先连接上它才能发出让它停止运行的命令（比如说，执行一条 mysqladmin shutdown 命令）。那么，哪些情况会导致这种局面呢？

首先，MySQL root 口令或许已设置了你未知的数值。这可能是在改变该口令时发生的。例如，你在输入新的口令值时键入了一个未曾见过的控制字符，或者你干脆忘记了该口令。

其次，Unix 下，至 localhost 的连接是通过一个 Unix 区域套接字文件（例如/tmp/mysql.sock）完成的。如果该套接字文件已删除，本地客户将不能使用它连接。这在系统运行 cron 作业时可能发生，cron 作业将不时去除/tmp 中的临时文件。

如果不能连接的原因是 Unix 套接字文件已被去除，可以重新启动服务器重新得到该文件。（当进行备份时服务器重建套接字文件。）文中的手段是因为套接字文件已不在，不能使用它建立一个连接来让服务器关停，只得建立 TCP/IP 连接。为了进行这项工作，利用--protocol=tcp 选项或指定主机值为 127.0.0.1（而不是 localhost）来连接至本地服务器：

```
% mysqladmin -p -u root --protocol=tcp shutdown
% mysqladmin -p -u root -h 127.0.0.1 shutdown
```

127.0.0.1 是一个 IP 号（作为本地主机回送的接口），所以它是显式强制使用 TCP/IP 连接，而不是套接字连接。

如果 Unix 套接字文件已被 cron 作业删除，失去套接字的问题会重新发生，除非你改变 cron 作业，或者使用某处的套接字文件。可以指定一个不同的套接字文件，在全局性选项文件中命名。例如，如果 MySQL 数据库目录为/usr/local/mysql，可以通过把下列各项加至/etc/my.cnf 来使用其中的套接字文件：

```
[mysqld]
socket=/usr/local/mysql/mysql.sock

[client]
socket=/usr/local/mysql/mysql.sock
```

作了修改以后重新启动服务器，以便在新的地点创建套接字文件。必须为服务器和客户程序都指定 Unix 套接字文件路径名，以便它们都使用同一个套接字文件。如果仅为服务器设置了路径名，则客户程序仍将希望找到在老地方的套接字。可惜，该方法只能用于读选项文件的客户，有些第三方程序能做，但有些不能做。如果从开始重新编译了 MySQL，则可以重新配置该分配，以便使服务器和客户都能默认使用不同的路径名。这还会自动影响使用客户数据库的第三方程序，除非它们已与旧库表态连接，此时你必须重编译它们才能使用新库。

如果由于未能记住或不知道 root 口令而不能连接，需要重新获得该服务器的控制，才能再次设置口令。为了进行这项工作，执行下列步骤。

(1) 关停服务器。在 Unix 下，如果能在服务器主机上以 root 登录，就可以使用 kill 命令终止服务器。通过查看服务器 PID 文件（该文件通常位于数据目录中）或者使用 ps 命令，可以找出服务器进程 ID。然后给服务器发出 TERM 信号让服务器进程关停：

```
# kill -TERM PID
```

用这种方法，数据表和记录将正确更新。如果服务器受到干扰和对正常终止信号无响应，则可使用 kill -9 强行终止服务器：

```
# kill -9 PID
```

Kill -9 是最后不得已的手段，因为内存中可能有未更新的修改，而且有数据表不一致的危险。

在 Linux 下，ps 可以展示几个 mysqld“进程”。这些实际上是同一进程的线程，所以你能去除它们任意一个，直到全部去除。

如果使用 mysql_safe 启动服务器，将会监视该服务器是否异常终止。如果利用 kill-9、mysql_safe 终止服务器，将会使它立即重启。为了避免这点，要确定 mysqld_safe 进程的 PID，并在结束 mysqld 后首先结束该进程。

如果在 Windows 下以一个服务运行服务器，即使不知道任何口令而正常关停服务器，但需要使用服务管理程序或发出下列命令：

```
C:\> net stop MySQL
```

为了在 Windows 上强制终止服务器，可使用任务管理程序（Alt-Ctrl-Del）。这类同于 Unix 上的 kill -9，这是最后不得已的手段。

(2) 用--skip_grant_tables 选项重新启动服务器，禁止使用验证连接用的权限表格。因此，你可以不用口令，而用所有权限进行连接。然而，这会让服务器完全开放，以便其他人以同样的方法连接。只要连接上了，就立即发出 FLUSH PRIVILEGES 语句：

```
% mysql
mysql> FLUSH PRIVILEGES;
```

FLUSH 语句告知服务器要重新读取权限表，使用表格进行访问控制。你仍然保持连接，但服务器需要用权限表验证其他客户进行的任何连接。FLUSH 语句还重新启用 SET DASSWORD 语句，因为该语句在服务器不用权限表时被禁用。在重新载入该表后，可以用 SET DASSWORD 或 UPDATE 改变 root 口令，如 12.1 节中的那样。如：

```
mysql> SET PASSWORD FOR 'root'@'localhost' = PASSWORD('rootpass');
```

(3) 在修改 root 口令之后，关闭服务器并按正常过程重新启动它。现在应该可以使用新的口令以 root 用户的身份连接上它了。

如果你在 Unix 下用 kill -9 或在 Windows 下用任务管理程序强制终止服务器，这种突然关停的操作将让服务器无法更新任何未保存磁盘的修改。为了处理由于这种关停可能产生的问题，比较好的主意是使服务器启用自动恢复的能力。请参阅 14.2.1 节的内容。

12.3 对 MySQL 服务器的连接监听情况进行控制

MySQL 服务器能够在多种网络接口上监听来自客户的连接请求，而我们可以像下面这样加以控制。

- 在所有的平台上，MySQL服务器都监听着供TCP/IP连接使用的网络接口，除非你在启动它时给出了--skip-networking选项。MySQL服务器使用的默认端口号是3306，你可以用--port选项另行指定一个不同的端口号。如果你的MySQL服务器主机有一个以上的IP地址，你还可以用--bind-address选项对服务器在监听客户连接时使用的IP地址进行设定。
- 在Unix系统上，服务器还将在一个Unix域套接字文件上监听有没有本地客户正在以localhost为主机名或是使用--protocol=socket选项尝试建立连接。MySQL服务器对套接字的这种用法是不能禁用的。默认的套接字文件通常是/tmp/mysql.sock，但包含MySQL的操作系统发行版本往往会使用一个不同的位置。也可以用--socket选项明确地给出套接字文件的路径名。
- 在支持命名管道的Windows服务器上，命名管道连接在默认的情况下是被禁用的。如果想启用这个功能，需要使用--enable-named-pipe选项来启动服务器。这将使本地客户可以通过给出--protocol=pipe选项或是通过连接主机名“.”的办法，从命名管道连接到服务器。在默认的情况下，命名管道的名字是MySQL（不区分字母的大小写），但你可以用--socket选项另外指定一个不同的命名管道名称。
- 在Windows系统上，MySQL还支持共享内存连接，但这种支持是默认禁用的。必须以--shared-memory选项启动服务器才能启用它。在启用之后，它将成为本地客户的默认连接协议。本地客户还可以使用--protocol=memory选项来明确地表明“我想使用共享内存”。在默认的情况下，共享内存的名字是MYSQL（区分字母的大小写），但你可以用--shared-memory-base-name选项另行指定一个不同的名字。

即使默认的本地连接协议不是 TCP/IP，客户也可以通过使用 127.0.0.1 作为服务器主机名来明确地表明“我想使用 TCP/IP 协议连接本地服务器”。那是 TCP/IP 回送接口的地址。强行建立 TCP/IP

连接的另一个办法是使用--protocol=tcp 选项。

如果只运行了一个服务器，常见做法是让服务器使用它的默认网络设置。如果运行着多个服务器，就必须保证不同的服务器有各自专用的网络参数，详见 12.10 节里的讨论。

以上介绍只适用于运行在客户/服务器环境里的独立的服务器。被链接到应用程序里的嵌入式服务器与其客户端之间的通信是通过一个内部链路实现的，根本不需要去监听任何外部的网络接口，所以这里的讨论不适用于嵌入式服务器。

12.4　管理 MySQL 用户账户

MySQL 管理员应该知道如何设置 MySQL 用户账户，即指定哪些用户可以连接到 MySQL 服务器，他们都可以从哪些地方连接，连接上服务器后又能做什么，等等。这些信息分别存放在 mysql 数据库中的几个权限表里，对 mysql 数据库的操作主要通过下面这些账户管理 SQL 语句来完成。

- CREATE USER、DROP USER和RENAME USER。这3条语句分别用来创建、删除和重新命名MySQL账户。
- GRANT。为某给定MySQL账户分配权限（如果账户不存在，则先创建之）。
- REVOKE。撤销某给定MySQL账户的权限。
- SET PASSWORD。为某给定MySQL账户设置口令。
- SHOW GRANTS。显示某给定MySQL账户当前拥有的全部权限。

这些账户管理语句将影响到 mysql 数据库里的以下几个权限表（见表 12-2）。

表 12-2　MySQL 的权限数据表

权　限　表	权限表的内容
user	可连接到服务器的用户和他们的全局级权限
db	数据库级权限
tables_priv	数据表级权限
columns_priv	数据列级权限
procs_priv	与存储例程有关的权限

还有一个名为 host 的权限表，但因为它不受这些账户管理语句的影响，并已经有点儿过时了，这里就不讨论了。

在使用 CREATE USER 语句时，必须给出一个由用户名和主机名构成的账户名，如果你愿意，还可以同时给该账户设置一个口令。服务器将在 user 数据表里为这个新账户创建一个数据行。这对 GRANT 语句来说（如果给定账户尚不存在的话）也是如此。在使用 GRANT 语句时，如果 GRANT 语句为账户设定了全局级权限(管理权限或者应用于所有数据库的权限)，它们将被记录到 user 权限表里。如果 GRANT 语句设定的权限只适用于某个给定的数据库、数据表、数据列或存储例程，它们将被分别记录到 db、tables_priv、columns_priv 和 procs_priv 权限表里。REVOKE 语句从权限数据表删除有关的权限信息，DROP USER 语句将从各表里把与给定账户相关的数据行全部删掉。

权限表的内容也可以通过 INSERT 和 UPDATE 等 SQL 语句来直接处理。不过，作为权限表的前端，GRANT 和 REVOKE 等账户管理语句让用户账户的管理工作变得更容易。它们在概念上更清晰，使用起来也更容易，你只要描述你想要进行的访问修改，MySQL 服务器就会自动地把你的请求映射为正确的权限表操作。不过，虽说使用 GRANT 和 REVOKE 语句的做法要比直接修改权限表更简明，我仍建议

大家把第 13 章的内容作为本节的补充材料。第 13 章对权限表的详细介绍能帮助大家更好地掌握这些账户管理语句的工作原理。

在接下来的几节里，我们将向大家介绍如何创建和删除 MySQL 用户账户，如何授权和撤销权限以及如何改变口令或重新设置丢失的口令。

说明 MySQL 的某些版本引入了新的权限，它们改变了权限表的内部结构。在你们第一次把 MySQL 安装到机器上时，安装过程将按照正被安装的 MySQL 版本所知道“最新的”结构来创建权限表。如果把 MySQL 升级到一个新版本，千万不要忘记运行 mysql_update 命令把那些权限表也升级到新版本的结构。

mysql_update 命令需要以 root 用户的身份连接到本地服务器才能运行，所以在使用它时必须给出正确的口令，如下所示：

```
% mysql_update --password=rootpass
```

12.4.1 高级MySQL账户管理操作

所谓高级 MySQL 账户管理操作指的是以下 3 条语句进行的操作。

- CREATE USER：创建一个新账户并为它设置一个口令（可选）：

  ```
  CREATE USER account [IDENTIFIED BY 'password' ];
  ```

  CREATE USER 语句只创建账户，不分配权限，权限分配工作由 GRANT 语句完成。
- DROP USER：删除一个现有的账户和与该账户相关联的全部权限：

  ```
  DROP USER account;
  ```

  DROP USER 语句不删除给定（已删除）账户可以访问的任何数据库和其中的数据对象。
- RENAME USER：改变现有账户的名字：

  ```
  RENAME USER from_account TO to_account;
  ```

你必须具备全局级 CREATE USER 权限才能同时使用这 3 条语句。否则，你必须具备 mysql 数据表的 INSERT、DELETE 或 PDATE 权限才能分别使用 CREATE USER、DROP USER 或 RENAME USER 语句。

在创建新账户时，可以通过回答下面这几个简单的问题来构造出相应的 CREATE USER 语句。

- 新用户的用户名是什么？
- 新用户将从哪一台或哪几台主机连接服务器？
- 新用户的口令是什么？

前两个问题用来确定 CREATE USER 语句里的 account 值，第三个问题用来确定 IDENTIFIED BY 子句里的口令。account 值必须遵守本节中第 1 小节将给出的规则。IDENTIFIED BY 子句里的 password 值必须是未经编码的口令文本，CREATE USER 语句将替你为口令编码，你不需要像在 SET PASSWORD 语句里那样使用 PASSWOR() 函数对它编码。如果你没有给出 IDENTIFIED BY 子句，新账户将没有任何口令，这是一种应该避免的安全隐患。

1. 指定账户名

CREATE USER 等账户管理语句中的 account 值由按照'user_name'@'host_name'格式给出的一个用户名和一个主机名构成。MySQL 不仅要求你必须指定谁能连接，还必须指定从什么地方连接。这意味着即便两个用户有着相同的名字，如果他们将从不同地点去连接服务器的话，你就要为他们各

自创建一个账户，MySQL 也知道该如何区别对待这两位用户，让你可以为他们授予相应的权限。服务器将在 user 权限表里为新账户创建一个数据行，并把新账户的 user_name 和 host_hame 值保存到该数据行的 User 和 Host 数据列里，也可保存到与这个新账户有关的其他权限表里。

你的 MySQL 用户名是你在连接服务器时用来表明自己身份的名字，它与 Unix 登录名或者 Windows 登录名没有任何必然联系。在 Unix 系统上，如果你在连接服务器时没有明确地给出一个 MySQL 用户名，程序将默认使用你的登录名作为你的 MySQL 用户名，但这只是一种约定俗成的做法而已。类似地，MySQL 把 root 用作其超级用户的名字也不是出于什么特殊的考虑。你完全可以把权限表里的这个名字改为 superduper、然后再以 superduper 作为你的用户名去连接 MySQL 服务器，并去进行各种必须具备超级用户权限才能进行的操作。

通过选择一个 account 值，就可以限制某个用户从一组指定的主机去连接 MySQL 服务器。作为一种极端的情况，如果知道某个用户只会从某一台主机去连接 MySQL 服务器，完全可以限制该用户只能使用那台主机进行连接：

```
CREATE USER 'boris'@'localhost' IDENTIFIED BY 'frost';
CREATE USER 'fred'@'ares.mars.net' IDENTIFIED BY 'steam';
```

请记住，主机名部分指的是客户将从哪些主机去连接服务器，不是客户连接到哪些服务器主机（除非二者相同）。

最为严格的访问控制形式是只允许用户从一台主机连接服务器。在另一种极端的情况下，你的用户因经常出差而需要从全球各地许多主机去连接服务器。假设这位用户的用户名是 max，下面这条语句将允许他从任何地方连接服务器：

```
CREATE USER 'max'@'%' IDENTIFIED BY 'mist';
```

“%”字符是一个通配符，它在这里的含义与它在 LIKE 模式匹配操作中的含义相同。因此，把主机名设置为“%”就表示“任意一台主机”。这种用户设置最为方便，但对系统却最不安全。（这种设置往往会给你带来很多令人头疼的问题，我们将在 13.2.3 节探讨原因。）

另一个 LIKE 通配符（减号）也可以用在主机名部分里，它可以匹配任意单个字符。

如果需要在名字里用到“%”或“-”字符本身，必须在它们的前面加上一个反斜线字符进行转义。

在两个极端之间，你通常需要允许用户从一组有限的主机连接服务器。比如说，如果你想让用户 mary 能够从 snake.net 域中的任意一台主机去连接服务器，就需要把主机名设置为'%.snake.net'：

```
CREATE USER 'mary'@'%.snake.net' IDENTIFIED BY 'fog';
```

如果你愿意，account 值的主机部分还允许以 IP 地址而不是主机名的形式给出。IP 地址既可以完全是文字，也可以包含模式匹配字符，还可以使用 IP 掩码来表明 IP 地址中的哪些位可用于网络编号，如下所示：

```
CREATE USER 'joe'@'192.168.128.3' IDENTIFIED BY 'water';
CREATE USER 'ardis'@'192.168.128.%' IDENTIFIED BY 'snow';
CREATE USER 'rex'@'192.168.128.0/255.255.255.0' IDENTIFIED BY 'ice';
```

上面第一条语句表明用户只能从 IP 地址是 192.168.128.3 的那台主机去连接服务器。第二条语句给出的是 C 类子网 192.168.128 中的一个 IP 地址范围。在第三条语句里，192.168.128.0/255.255.255.0 中的掩码表明前 24 位是有效的，即允许用户从其 IP 地址的前 24 位等于 192.168.128 的任何一台主机去连接 MySQL 服务器。掩码值必须是 255.0.0.0、255.255.0.0、255.255.255.0

或 255.255.255.255。

在账户名里使用 localhost 作为主机名将允许用户从本地主机来连接服务器，这又分为以下几种情况。

- 在Unix系统上，用户可以使用localhost或127.0.0.1为主机名来连接服务器。如果主机名是localhost，将使用Unix套接字文件建立连接；如果主机名是127.0.0.1，将使用本地主机的IP回送接口建立TCP/IP连接。
- 在Windows系统上，用户可以使用localhost或127.0.0.1为主机名来连接服务器。这两种情况都将使用TCP/IP协议来建立连接，但如果服务器支持共享内存连接的话，以localhost作为主机名将默认使用共享内存来建立连接。此外，如果服务器支持命名管道连接，用户还可以使用“.”作为主机名通过命名管道建立连接。

如果 account 值的用户名或主机名部分可以用作一个无须转义的标识符，就不必用引号把它括起来；如果它包含“-”或“%”等特殊字符，就必须用引号把它括起来。比如说，如果账户名是 boris@localhost，它的用户名和主机名部分都是无须转义的合法标识符，不需要加上引号。不过，坚持使用引号最保险，本书中的示例都遵守这条规则。在给用户名和主机名加引号的时候，既可以使用字符串引号字符，也可以使用标识符引号字符。需要特别注意的是，用户名和主机名必须分别放在两对单引号里，即必须写成'boris'@'localhost'，不能写成'boris@localhost'。

如果你在创建账户时没有给出它的主机名部分，在效果上就等同于把主机名部分设置为“%”。也就是说，'max'和'max'@'%'是等价的 account 值。这意味着如果你想把某个账户设置为'boris'@'localhost'却误写成了'boris@localhost'，MySQL 仍认为它是一个合法的账户。MySQL 将把'boris@localhost'整个地解释为该账户的用户名部分，并给它加上一个'%'作为其默认的主机名部分，最终得到的账户名将是'boris@localhost'@'%'。要想避免这种错误，就一定要给账户名里的用户名和主机名分别加上单引号。

2. 如何在账户名里指定本地主机名

使用服务器的主机名而不是 localhost 去连接服务器经常会遇到一些麻烦。这往往是因为你在权限表里写出的主机名与你的域名解析器所报告出来的不一致。我们不妨假设主机的完全限定名是 cobra.snake.net。那么，如果域名解析器报告出来的是一个不完整主机名（如 cobra），而权限表里的是完整名称（或正好相反），就会造成不匹配。

如果你想知道在你的系统上会不会发生这种问题，可以使用指定主机名称的-h 选项连接一下本地服务器试试，比如：

```
% mysql -h cobra.snake.net
```

然后查看一下服务器的常规日志文件。常规日志记载你的这次连接时写的主机名是什么？是完整的还是不完整的？常规日志里记载的本地主机名是什么格式，你就应该使用什么格式来给出有关账户的本地主机名部分。

12.4.2 对账户授权

对账户授权需要使用 GRANT 语句，下面是 GRANT 语句的语法：

```
GRANT privileges (columns)
  ON what
  TO account [IDENTIFIED BY 'password']
```

```
[REQUIRE encryption requirements]
[WITH grant or resource management options];
```

如果在 GRANT 语句里给出的账户已经存在，GRANT 语句将改变它的权限。如果那个账户尚不存在，GRANT 语句将先创建它，再把给定的权限分配给它。启用 NO_AUTO_CREATE_USERSQL 模式可以防止 GRANT 语句创建出一个没有任何口令的新账户（因为这是一种严重的安全隐患）。该模式是从 MySQL 5.0.2 版本开始引入的，在启用该模式之后，不带 IDENTIFIED BY 子句的 GRANT 语句将不能创建账户。

有几个子句是可选的，如果用不着，就不必把它们写出来。下面是 GRANT 语句最为常用的几个语法元素。

- privileges，授予账户的权限。比如说，SELECT权限将允许用户发出SELECT语句，SHUTDOWN权限将允许用户关停服务器。你可以一次授予多项权限，它们之间要用逗号隔开。
- columns，权限将作用于哪些数据列。如果需要列举多个数据列，必须把它们的名字用逗号隔开。这个子句是可选的，只有在设置数据列级权限时才必须给出。数据列的名单必须紧跟在每项相关权限的后面。
- what，权限的级别。最高级别是全局级，即给定权限将作用于所有的数据库和所有的数据表，你可以把全局级权限视为超级用户权限。权限还可以被设定为数据库级、数据表级、数据列级（当你给出一条columns子句时）或存储例程级。
- account，被授予权限的账户。account值的格式是'user_name'@'host_name'，详见12.4.1节中第1小节里的讨论。
- password，账户的口令。这个子句是可选的，如果账户已经存在并且有一个口令，就不必写出这个子句。如果为现有账户的权限时给出了IDENTIFIED BY子句，该账户现有的口令将被替换为你给出的新口令。类似于CREATE USER语句的情况，如果在GRANT语句里使用了IDENTIFIED BY子句，该子句里的password值应该是口令的明文形式。GRANT语句将自动为之加密，千万不要使用PASSWORD()函数。

REQUIRE 和 WITH 子句都是可选的。REQUIRE 子句的用途是对必须经由 SSL 进行安全连接的账户进行设置。WITH 子句用来授予 GRANT OPTION 权限，这个权限允许账户把自己的权限转授给他用户。WITH 子句还可以用来设置资源管理选项，这使 MySQL 管理员可以限制账户每小时可以进行多少次连接，可以执行多少条语句。这些选项有助于避免某些账户过度占用服务器。

在对 MySQL 账户授权时，可以通过回答下面这几个简单的问题来构造出相应的 GRANT 语句。

- 这个账户应执行哪些访问操作？或者说，这位用户应该拥有什么权限级别？这些权限又将作用于哪些对象？
- 是否需要使用安全连接？
- 是否要向这位用户授予一些管理员权限？
- 是否需要限制这位用户的资源占用量？

下面，我们将通过一些示例来回答这些问题，并演示如何使用 GRANT 语句的各种子句。

1. 确定账户的权限

可以授予用户的权限有很多种。我们把这些权限分别汇总在表 12-3、表 12-4 和表 12-5 里，并将在第 13 章对它们的用途和它们与各权限表的关系做详细的介绍。

表 12-3　数据库管理权限

权　限　名	该权限所允许的操作
CREATE USER	使用高级账户管理语句
FILE	读、写 MySQL 服务器主机上的文件
GRANT OPTION	把本账户的权限授予其他账户
PROCESS	查看在服务器里运行的线程的信息
RELOAD	重新加载权限数据表或者更新日志及缓存
REPLICATION CLIENT	查询主/从服务器的运行地点
REPLICATION SLAVE	以复制的从服务器运行
SHOW DATABASE	用 SHOW DATABASE 语句查看全体数据库的名字
SHUTDOWN	关闭服务器
SUPER	用 KILL 命令终止线程以及进行其他超级用户操作

表 12-4　数据库对象操作权限

权　限　名	该权限所允许的操作
ALTER	更改数据表和索引的定义
ALTER ROUTINE	更改或删除存储函数和存储过程
CREATE	创建数据库和数据表
CREATE ROUTINE	创建存储函数和存储过程
CREATE TEMPORARY TABLES	用 TEMPORARY 关键字创建临时数据表
CREATE VIEW	创建视图
DELETE	删除数据表里的现有数据行
DROP	删除数据库、数据表和其他对象
EVENT	为事件调度程序创建、删除或修改各种事件
EXECUTE	执行存储函数和存储过程
INDEX	创建或者删除索引
INSERT	往数据表里插入新数据行
LOCK TABLES	用 LOCK TABLES 语句明确地锁定数据表
REFERENCE	未使用（保留供今后使用）
SELECT	检索数据表里的数据行
SHOW VIEW	用 SHOW CREATE VIEW 语句查看视图的定义
TRIGGER	创建或者删除触发器
UPDATE	修改数据行

表 12-5　其他权限

权　限　名	该权限所允许的操作
ALL [PRIVILEGES]	所有操作（但不包括 GRANT 权限）
USAGE	一个特殊的“无权限”权限

表 12-3 里的权限是数据库管理权限。这类权限控制着服务器的运行情况，所以很少被授予普通用户。（比如说，用来关闭服务器的 SHUTDOWN 权限肯定不应该授予普通用户。）表 12-4 里的权限作用于

数据库、数据表、数据列和存储例程，它们控制着用户对服务器所管理的数据的访问。表 12-5 里的权限有特殊的用途：ALL 代表“所有权限”（但不包括 GRANT OPTON 权限），USAGE 代表“无权限”，意思是“创建一个账户，但不给它授予任何权限”。USAGE 权限的主要用途是在不改变当前权限的前提下修改该账户与访问权限没有关系的项目（详见下一小节）。

CREATE VIEW 和 SHOW VIEW 权限是从 MySQL 5.0.1 版开始新增加的。ALTER ROUTINE、CREATE ROUTINE 和 CREATE USER 是从 MySQL 5.0.3 版开始引入的，EXECUTE 权限也是从这个版本开始可使用的。EVENT 和 TRIGGER 权限是从 MySQL 5.1.6 版开始引入的。（在 MySQL 5.1.6 版之前，与触发器有关的操作需要 SUPER 权限才能进行，那时还没有 TRIGGER 权限）。

要想把权限授予其他账户，你本人必须具备该权限，而且你必须还具备 GRANT OPTION 权限。

MySQL 允许在数据库系统全局、数据库、数据表、数据列等多种级别上进行授权。权限的级别由 ON 子句控制，如表 12-6 所示。

表 12-6　权限级别限定符

权限限定符	有关权限的作用范围
ON *.*	全局级权限，其作用范围是所有数据库及其中的所有对象
ON *	如果没有指定默认的数据库，这是全局级权限；否则，是默认数据库上的数据库级权限
ON *db_name*.*	数据库级权限，其作用范围是指定数据库里的所有对象
ON *db_name.tbl_name*	数据表级权限，其作用范围是指定数据表里的所有数据列
ON *tbl_name*	数据表级权限，其作用范围是默认数据库中指定数据表里的所有数据列
ON *db_name.routine_name*	存储例程权限，其作用范围是指定数据库里的指定例程

从 MySQL 5.0.6 版本开始，为了避免歧义，可以使用 TABLE、FUNCTION 或 PROCEDURE 关键字明确地对有关权限的作用对象作出限定（如 ON TABLE mydb.mytbl 或 ON FUNCTION mydb.myfunc）。

USAGE 权限只能在全局级别授予（即必须与 ON *.*配合使用）。

对于数据表级权限限定符，还可以用一个（column）子句把紧随其后的数据列级权限授予给定的数据列，稍后的例子演示了这么做的语法。

ALL 权限限定符将把给定级别的所有权限授予给定账户。比如说，在全局级别，它将把全部的权限授予给定账户；在数据表级，它将只把适用于数据表的权限授予给定账户。ALL 只能用来授予全局级、数据库级、数据表级或例程级权限。对于数据列级权限，你必须明确地列出你打算授予的每一种权限。

全局级权限的作用范围最大，它们将作用于任何数据库。比如说，下面这些语句将创建出一个能做任何事情（包括对其他用户进行授权）的超级用户账户：

```
CREATE USER 'ethel'@'localhost' IDENTIFIED BY 'coffee';
GRANT ALL ON *.* TO 'ethel'@'localhost' WITH GRANT OPTION;
```

ON *.*子句的意思是“所有的数据库和它们里面的所有对象”。出于安全考虑，上面这个例子所创建的账户只能从本地主机连接 MySQL 服务器。限制 MySQL 超级用户只能从某几台主机去连接服务器是一种极其明智的做法，因为要想盗取 MySQL 超级用户的口令，黑客们只能从这几台主机入手。

表 12-3 里的权限（但不包括 GRANT 权限）是专为数据库管理员准备的，只能通过全局级权限说明符 ON *.*来进行授予。比如说，RELOAD 权限将允许你使用 FLUSH 语句，下面这些语句将创建出一

位名为 flush 的用户，这位用户除了能发出 FLUSH 语句外其他什么事情都不能做：

```
CREATE USER 'flush'@'localhost' IDENTIFIED BY 'flushpass';
GRANT RELOAD ON *.* TO 'flush'@'localhost';
```

这类 MySQL 账户特别适合用来编写一些需要完成某种管理性操作的脚本。比如说，在对日志文件进行维护更新日志（请参阅 12.5.7 节）。

数据库级权限将作用于一个特定的数据库和它里面的所有对象。这个级别的权限需要使用 ON db-name.*子句来授予：

```
CREATE USER 'bill'@'racer.snake.net' IDENTIFIED BY 'rock';
GRANT ALL ON sampdb.* TO 'bill'@'racer.snake.net';

CREATE USER 'reader'@'%' IDENTIFIED BY 'dirt';
GRANT SELECT ON menagerie.* TO 'reader'@'%';
```

上面的第一组语句创建了一个名为 bill 的用户，当他从名为 racer.snake.net 的主机连接到服务器时，他将获得 sampdb 数据库里的所有数据表的所有权限。第二组语句创建了一个名为 reader 的用户，他可以从任何主机连接服务器并访问 menagerie 数据库里的任何一个数据表，但只能通过 SELECT 语句去访问。换句话说，reader 是一个“只读”用户。

可以在 GRANT 语句里同时列出多个权限，但必须用逗号隔开。比如说，如果想让某位用户能读取和修改 sampdb 数据库里的现有数据表的内容，但不能创建新数据表或删除现有数据表，就不能把 ALL 权限授给这位用户。应该像下面这样列出具体的权限：

```
CREATE USER 'jennie'@'%' IDENTIFIED BY 'boron';
GRANT SELECT,INSERT,DELETE,UPDATE ON sampdb.* TO 'jennie'@'%';
```

要想在数据库级别下实现更细致的访问控制，就要在各表，甚至是在各列上进行授权操作。如果想让表中某些数据列对某位用户不可见，或者想让这位用户只能修改其中的某几个数据列，就需要用到数据列级权限。假设你是“美国历史研究会”的秘书，现在有位志愿者要来帮你做些杂事。这是件好事，但你决定在开始时只把（包含会员信息的）member 数据表上的只读权限，和该数据表里的 expiration 和地址数据列上的 UPDATE 权限授予这位新助手。这样，当某位会员申请延长其会员资格或是变更地址时，你就可以放心地让这位新助手替你去完成那些工作了。下面这些语句将创建出一个符合上述要求的 MySQL 账户：

```
CREATE USER 'assistant'@'localhost' IDENTIFIED BY 'officehelp';
GRANT SELECT, UPDATE (expiration,street,city,state,zip)
  ON sampdb.member TO 'assistant'@'localhost';
```

GRANT 语句将把对整个 member 数据表的读权限授予那位新助手（因为在 SELECT 后面没有列出任何数据列），还将把 UPDATE 权限授予他，但其作用范围仅限于在紧随 UPDATE 之后的括号里的那些数据列。

如果想用一条 GRANT 语句授予多种数据列级的权限，必须在每种权限的名字后面分别提供一个放在括号里的数据列名单。

在给 GRANT 语句里的数据库、数据表或数据列的名字加反引号时，必须把它们当做标识符（而不是字符串）来对待，如下所示：

```
GRANT SELECT, UPDATE (`expiration`,`street`,`city`,`state`,`zip`)
  ON `sampdb`.`member` TO 'assistant'@'localhost';
```

权限表里的数据行不“追随”数据库对象的重新命名操作。比如说，如果你更改了数据表或数据列的名字，与该数据表或数据列关联的所有权限将全部失效。

2. 使用“无权限”的USAGE权限

USAGE 权限说明符的含义是“没有权限”。第一眼看去，它好像没什么用，其实不然。当你想在保持其现有权限的情况下去修改某个账户与权限无关的特性（如用户名、主机名等）时，就需要用到 USAGE 权限。USAGE 的用法是：授予全局级 USAGE 权限，指定账户名，给出该账户与权限无关的特性值。比如说，如果你想在不影响账户权限的前提下改变该账户的口令，或要求用户必须使用 SSL 连接，或者想对该账户限制连接，就要使用下面几条语句：

```
GRANT USAGE ON *.* TO account IDENTIFIED BY 'new_password';
GRANT USAGE ON *.* TO account REQUIRE SSL;
GRANT USAGE ON *.* TO account WITH MAX_CONNECTIONS_PER_HOUR 10;
```

3. 要求账户必须使用安全连接

MySQL 允许使用 SSL（Security Socket Layer，安全套接字层）协议来建立安全连接，它将对服务器和客户程序之间的数据流进行加密，使它们不再以明文形式传输。此外，X509 可用于让客户程序在 SSL 连接上提供身份验证信息。安全连接给信息增加了一层安全屏障，但因为需要进行加密和解密运算，所以会加重 CPU 的负担。

与安全连接有关的选项要用 REQUIRE 子句来设置。REQUIRE SSL 表示要求用户必须通过 SSL 连接 MySQL 服务器，但对其使用的安全连接类型不做具体要求：

```
CREATE USER 'eladio'@'%.snake.net' IDENTIFIED BY 'flint';
GRANT ALL ON sampdb.* TO 'eladio'@'%.snake.net' REQUIRE SSL;
```

如果还想严格一些，可以要求客户提供一份合法的 X509 证书：

```
GRANT ALL ON sampdb.* TO 'eladio'@'%.snake.net' REQUIRE X509;
```

REQUIRE X509 只要求客户提供的证书必须是合法的，对证书的内容没有任何限制。如果还想更严格，可以要求客户的X509证书必须具备某些特征。这些特征由REQUIRE子句中的ISSURE或SUBJECT选项设定。ISSUER 和 SUBJECT 分别代表证书的签发者和持有者。比如说，在 sampdb 发行版本中的 ssl 目录里有一份客户身份证书文件 client-cert.pem，可以用来测试 SSL 连接。这份证书的签发者和持有者可以用 openssl 命令显示出来：

```
% openssl x509 -issuer -subject -noout -in client-cert.pem
issuer= /C=US/ST=WI/L=Madison/O=sampdb/OU=CA/CN=sampdb
subject= /C=US/ST=WI/L=Madison/O=sampdb/OU=client/CN=sampdb
```

下面这条 GRANT 语句所创建的账户要求客户提供的证书必须匹配签发者和持有者：

```
GRANT ALL ON sampdb.* TO 'eladio'@'%.snake.net'
  REQUIRE ISSUER '/C=US/ST=WI/L=Madison/O=sampdb/OU=CA/CN=sampdb'
  AND SUBJECT '/C=US/ST=WI/L=Madison/O=sampdb/OU=client/CN=sampdb';
```

还可以使用 REQUIRE 子句要求连接必须使用某种特定的密码来加密：

```
GRANT ALL ON sampdb.* TO 'eladio'@'%.snake.net'
  REQUIRE CIPHER 'DHE-RSA-AES256-SHA';
```

要想明确地表明无须使用安全连接，请使用 REQUIRE NONE 子句。这是创建新账户时的默认设置，但可以用来去掉加在某给定账户上的现有 SSL 限制。

在使用 REQUIRE 子句时，还需要注意以下几个问题。

- 发出一条要求账户必须使用安全连接的GRANT语句时仅表明你对这个账户作出了一项限制，这并不能让人们使用该账户建立安全连接。要想建立安全连接，就必须配置MySQL，使之支持SSL，并且以某种方式启动服务器和客户程序，具体做法参见13.3节。
- 如果设定了某个账户必须使用SSL来建立连接，但服务器和客户程序都不支持SSL，那个账户将无法使用。
- REQUIRE子句只是用来规定账户是否必须使用安全连接。只要把服务器和客户程序都配置成支持SSL，即使没有要求，任何用户都可以使用安全连接。
- 如果某个账户在连接服务器时不需要经过外部网络，用REQUIRE子句就没有实际的意义。这类连接不可能被搜索，所以要求它们必须是加密连接只会加重计算负担而不会让你得到任何好处。这类账户包括只通过一个UNIX套接字文件命名管道或共享内存连接MySQL服务器的账户，连接到IP地址127.0.0.1（主机回送接口）的账户。这些连接所使用的接口都由主机内部处理，信息不会泄露到外部网络。

4. 把管理权限授予账户

如果给出 WITH GRANT OPTION 子句，账户将可以把它自己的权限转授给其他账户，但你本人必须具备 GRANT OPTION 权限才能使用这个子句。

为账户授予 GRANT OPTION 权限的一个原因是，让数据库的拥有者能控制对具备该数据库的访问，这就需要把该数据库上的全部权限（包括 GRANT OPTION 权限）授予这位用户。比如说，如果想让用户 alicia 能够从 big-corp.com 域里的任何一台主机去连接服务器，并拥有 sales 数据库中所有数据表的管理权限，就应该创建一个如下所示的账户：

```
CREATE USER 'alicia'@'%.big-corp.com' IDENTIFIED BY 'shale';
GRANT ALL ON sales.* TO 'alicia'@'%.big-corp.com' WITH GRANT OPTION;
```

事实上，利用 WITH GRANT OPTION 子句可让你把自己的权限转授给其他用户。要注意，两个拥有 GRANT OPTION 权限的用户可以将自己的权限授予对方。比如说，假设你只向用户 A 授予了 SELECT 权限，但用户 B 拥有 GRANT OPTION 权限、SELECT 权限和其他几种权限，那么用户 B 就能使用户 A 变得更“强大”。

GRANT OPTION 权限的另一种授予方法是把它直接写在 GRANT 语句的开头部分，如下所示：

```
GRANT GRANT OPTION ON sales.* TO 'alicia'@'%.big-corp.com';
```

但下面这条语句却是错误的：

```
GRANT ALL,GRANT OPTION ON sales.* TO 'alicia'@'%.big-corp.com';
```

在 GRANT 语句里，ALL 只能单独出现，不能和其他的权限说明符列在一起。

GRANT OPTION 权限将作用于等于或低于它本身所处的权限级别的所有权限，而不是所有的权限。如果某个账户拥有某给定级别的 GRANT OPTION 权限，他将可以把自己拥有的该级别和该级别以下的任何权限转授给其他用户。你无法限定账户在可以转授一些权限的同时却不能转授同一级别的其他权限。

5. 限制账户的资源占用量

MySQL 的授权机制可以让你限制账户每小时可以连接多少次服务器，每小时可以发出或修改多少语句。这些限值要用 WITH 子句来设置。下面这些语句把 sampdb 数据库上的全部权限都授予了

'spike '@'localhost'用户，但只允许他每小时最多连接 10 次、每小时最多发出 200 条语句命令（最多只能有 50 条是修改命令）：

```
CREATE USER 'spike'@'localhost' IDENTIFIED BY 'pyrite';
GRANT ALL ON sampdb.* TO 'spike'@'localhost'
  WITH MAX_CONNECTIONS_PER_HOUR 10 MAX_QUERIES_PER_HOUR 200
  MAX_UPDATES_PER_HOUR 50;
```

每个选项的默认值都是零，其含义是“没有上限”。这意味着如果你曾经给某个账户设置过资源占用上限，可以通过把上限值设置为零来取消该限制。比如说，下面这条语句将取消刚才对 spike 用户作出的每小时最多只能连接 10 次的限制：

```
GRANT USAGE ON *.* TO 'spike'@'localhost'
  WITH MAX_CONNECTIONS_PER_HOUR 0;
```

用户不能通过同时建立多条连接避免这些限制，因为给定账户的全部连接是累加在一起的。

MySQL 从 5.0.3 版本开始增加了第四个资源管理选项，即 MAX_USER_CONNECTIONS。它用来为账户能够同时建立的连接个数设置上限值。如果这个上限值是零（默认值），则最大连接个数将由系统变量 max_user_connections 控制；如果它是一个非零值，那个值将是能够同时建立的连接的最大个数。

资源管理选项在 WITH 子句中的次序无关紧要。

如果管理员用户具备 RELOAD 权限，他就能通过发出一条 FLUSH USER_RESOURCES 语句对当前计数值复位，FLUSH PRIVILEGES 语句也能做到这一点。在计数值复位之后，那些资源占用量已经达到上限值的账户就能再次连接和发出语句了。这种复位还会发生在你用一条 GRANT 语句为该账户设置资源限制的时候。

12.4.3　查看账户的权限

账户所拥有的权限可以用 SHOW GRANTS 语句查看：

```
SHOW GRANTS FOR 'sampadm'@''localhost' ;
```

如果想查看你自己的权限，使用以下两条语句之一均可：

```
SHOW GRANTS;
SHOW GRANTS FOR CURRENT_USER();
```

12.4.4　撤销权限和删除用户

撤销某位用户的部分或全部权限要使用 REVOKE 语句。除了把 TO 换成了 FROM 以外，REVOKE 语句与 GRANT 语句在语法上非常相似，但 REVOKE 语句没有 IDENTIFIED BY、REQUIRE 和 WITH 子句：

```
REVOKE privileges [(columns)] ON what FROM account;
```

比如说，下面的 GRANT 语句为账户授予了 sampdb 数据库上的全部权限，但 REVOKE 语句只撤销了该账户对现有数据行进行删除和修改的权限：

```
GRANT ALL ON sampdb.* TO 'boris'@'localhost';
REVOKE DELETE,UPDATE ON sampdb.* FROM 'boris'@'localhost';
```

GRANT OPTION 权限不包括在 ALL 权限中，所以如果你在授予该权限后又想撤销它，就只能在

REVOKE 语句的 privileges 部分里把它明确地写出来：

```
REVOKE GRANT OPTION ON sales.* FROM 'alicia'@'%.big-corp.com';
```

要想撤销某个权限，你本人必须具备该权限，并且还必须具备 GRANT OPTION 权限。

下面这条语句将把给定账户在各个级别拥有的全部权限都撤销：

```
REVOKE ALL PRIVILEGES,GRANT OPTION FROM account;
```

请注意，在这个语法里没有 ON 子句。此外，必须具备全局级 CREATE USER 权限或是 mysql 数据库上的 UPDATE 权限才能执行这条语句。

当撤销账户在数据库、数据表、数据列或例程级别上的所有权限时，MySQL 将从 db、tables_priv、columns_priv 或 procs_priv 权限表里把与该账户相关的数据行删除掉。撤销某个账户的全部全局级权限将把该账户在 user 数据表里的行的权限数据列设置为'N'，但不会删除那个数据行。也就是说，REVOKE 语句不能彻底删除一个账户，这意味着那个用户仍能连接服务器。要想彻底删除某个账户，必须使用 DROP USER 语句，不能使用 REVOKE 语句（请参阅 12.4.1 节）。

矛盾的是，有些权限必须使用 GRANT 语句才能撤销。比如说，如果你指定账户必须使用 SSL 去连接，却没有用来撤销这项要求的 REVOKE 语法，所以你只能用一条如下所示的 GRANT 语句来做这件事：在全局级授予 USAGE 权限（保留该账户的现有权限），再用 REQUIRE NONE 子句表明不再要求它使用 SSL 来连接：

```
GRANT USAGE ON *.* TO account REQUIRE NONE;
```

类似地，你为用户设置的资源限制也无法用 REVOKE 语句来删除只能用包含 USAGE 的 GRANT 语句把限值设置为零（即“没有上限”）：

```
GRANT USAGE ON *.* TO account
  WITH MAX_CONNECTIONS_PER_HOUR 0 MAX_QUERIES_PER_HOUR 0
  MAX_UPDATES_PER_HOUR 0;
```

12.4.5 改变口令或重新设置丢失的口令

修改或重新设置账户口令的一种办法是使用 UPDATE 语句，为该账户的 User 表中的行标识 Host 值和 User 值，再重新加载权限表：

```
mysql> USE mysql;
mysql> UPDATE user SET Password=PASSWORD('silicon')
    -> WHERE User='boris' AND Host='localhost';
mysql> FLUSH PRIVILEGES;
```

更简单的办法是使用 SET PASSWORD 语句，只要按照其他账户管理语句里的格式写出账户名即可，权限表也用不着重新加载了：

```
mysql> SET PASSWORD FOR 'boris'@'localhost' = PASSWORD('silicon');
```

只要不是以匿名用户的身份建立连接的，你可以用 SET PASSWORD 语句来修改自己的口令。如果你想修改其他账户的口令，你本人必须具备 mysql 数据库上的 UPDATE 权限。

还有一种不常见的口令修改办法是使用包含 IDENTIFIED BY 子句的 GRANT USAGE 语句，直接写出口令文本即可，不要使用 PASSWORD()函数：

```
mysql> GRANT USAGE ON *.* TO 'boris'@'localhost' IDENTIFIED BY 'silicon';
```

当你因为忘了 root 口令而无法连接服务器时，就需要重新设置 root。这就产生了这样一个矛盾：你应该先用 root 口令连接上服务器才能修改 root 口令。如果不知道 root 口令，就只能先强行关停服务器，然后在不使用权限表验证的情况下重新启动服务器，这一过程的具体步骤见 12.2.5 节。

12.5　维护日志文件

MySQL 服务器有能力生成好几种日志。它们在诊断故障、改善服务器性能、建立复制机制和崩溃恢复等工作中很有用。在它开始运行的时候，MySQL 服务器会去检查它的启动选项看是否应该启用日志功能，如果是，则打开相应的日志文件。可以让服务器生成几种不同类型的日志。下面是对各日志的简要描述，接下来的几个小节将提供更多的细节。

- **出错日志（error log）**。这个日志记载着服务器启动和关闭的情况，还记载着关于故障或异常状况的消息。如果服务器无法启动，首先应该查看一下这个日志。在意外发生时，服务器会在结束运行前把一条消息写入出错日志以表明发生了什么问题。
- **常规查询日志**。该日志包括客户连接的记录、来自客户的SQL查询和其他各种的事件。这有助于监视服务器的活动：谁在连接，从何处连接和他们在做什么。当你想要确定客户发送至服务器的是什么查询时，这是最方便使用的日志，这对故障诊断或调试十分有用。
- **慢查询日志**。该日志的用途是为了改善性能，帮助你区别重写所需要的语句。服务器维护定义为“慢”查询（默认为10秒）的long-query-time变量。如果查询要花费好多秒，就认为是慢的，并记录在慢查询日志中。慢查询日志也用于记录不用索引的查询。
- **二进制日志（binary log）和二进制日志索引文件**。这个日志由一个或多个文件构成，里面记载着由UPDATE、DELETE、INSERT、CREATE TABLE、DROP TABLE、GRANT等语句完成的数据修改情况。二进制日志的内容是一些二进制编码的数据修改“事件”。二进制日志文件有一个配套的索引文件，里面列出了服务器上现有的二进制日志文件。

  二进制日志有两个基本用途。

  - 它可以配合数据库备份文件在系统发生崩溃后对数据表进行恢复。先从备份文件恢复数据库，然后使用 mysqlbinlog 工具把二进制日志的内容转换为文本语句。接下来，把上次备份后执行过的每一条数据修改语句依次馈入 mysql 程序执行，就可以把数据库的状态恢复到崩溃发生前的那一刻。
  - 在复制机制中，通过二进制日志把主服务器上发生的数据修改事件传输到从服务器去。
- **中继日志（relay log）和中继日志索引文件**。如果某个服务器是复制机制中的从服务器，它将维护着一个中继日志，里面记载着从主服务器接收的、目前尚未执行的数据修改事件。中继日志文件和二进制日志文件的格式是一样的，并且也有一个配套的索引文件列出了从服务器上现有的中继日志文件。

在所有这些日志当中，常规日志最适合用来监控服务器的运行状态。因此，在刚开始学习使用 MySQL 的起步阶段，建议大家在启用其他日志的时候别忘了把常规日志也加进去，等获得了一定的 MySQL 经验之后再关闭常规日志功能以减少硬盘空间的消耗。

在默认的情况下，每一个被启用的日志都将被写入数据目录中的某个文件（或某一组文件）。在 MySQL 5.1 系列版本里，还可以选择把某些日志写入其他的地点：出错日志还可以发送到 syslog，常规日志和慢查询日志还可以写到 mysql 数据库中的数据表里去。

除非系统管理员明确地提出了要求，否则 MySQL 服务器是不会主动地创建任何日志的，但这里有两个例外。

- 在Unix系统上，如果使用了mysqld_safe脚本来启动服务器，该脚本将创建出错日志并告诉服务器去使用它。
- 在Windows系统上，只要没有使用--console选项表明你想让诊断消息被发送到控制台窗口而不是被发送到一个文件，服务器就会创建出错日志。

日志由 mysqld 程序的部分启动选项控制。除二进制日志和中继日志以外，这些日志都是以可以直接阅读的文本格式写的。二进制或中继日志文件的内容可以使用 mysqlbinlog 实用工具程序来查看。

如果想启用日志功能，可以使用下表列出的那些选项。如果日志文件的名字是可选的（以方括号表示）而你又没有提供一个文件名，服务器将使用一个默认文件名把日志文件写到数据目录里。服务器将根据 MySQL 服务器主机的名字为日志文件构造一个默认文件名，这个主机名在接下来的讨论里用 HOSTNAME 代表。如果给出的日志文件名是一个相对路径，服务器将把它解释为一个相对于数据目录的名字。可以给出一个完整路径名把日志文件放到其他的子目录里去。如果某个日志文件不存在，服务器将自动创建该文件。但如果用来保存日志文件的子目录不存在，MySQL 服务器不会自动创建该子目录。因此，在启动 MySQL 服务器之前，一定要把必要的子目录全部创建好。

日志选项	由本选项启用的日志
--log-error [=*file_name*]	出错日志文件
--log [=*file_name*]	常规日志文件
--log-slow-queries [=*file_name*]	慢查询日志文件
--log-output [=*destination*]	常规/慢查询日志的存放地点
--log-bin [=*file_name*]	二进制日志文件
--log-bin-index =*file_name*	二进制日志索引文件
--log-relay [=*file_name*]	中继日志文件
-- relay - log -index =*file_name*	中继日志索引文件

可以在 mysqld 程序或 mysqld_safe 脚本的命令行上给出 MySQL 服务器的日志选项。不过，因为在每次启动 MySQL 服务器时给出的日志选项通常都是一样的，所以把它们放入某个选项文件里一个适当的选项组的情况更常见。通常的做法是把它们放入[mysqld]或[mysqld_safe]选项组，但它们并非必须如此。12.2.3 节对适用于服务器本身和服务器启动程序的选项组进行了详细的说明。

还有一些特殊用途的日志是由个别存储引擎管理的。ISAM 日志用于调试，它记载着对 MyISAM 数据表的修改，我以后不再提到它。如果启用了 InnoDB 和 Falcon 存储引擎，它们会创建一些日志供系统发生崩溃后的自动恢复功能使用。你不能控制这些存储引擎是否会生成它们的日志，但可以利用下表列出的选项让它们把日志写到你指定的地方。默认的写入地点是数据目录。

日志选项	用途
--innodb_log_group_home_dir = *dir_name*	InnoDB日志文件子目录
--falcon_serial_log_dir = *dir_name*	Falcon日志文件子目录

刷新日志

刷新日志将导致服务器先关闭再打开日志文件。这可以通过执行一个 mysqladmin flush-logs 命令或一条 FLUSH LOGS 语句来完成。在 Unix 系统上，向 MySQL 服务器发送一个 SIGHUP 信号也是让它刷新日志的意思。mysqladmin refresh 命令将刷新日志，但它还会做一些其他的事情，如刷新数据表缓存，所以如果只想刷新日志的话，用它来做这件事就有点儿大材小用了。

二进制和中继日志文件是按编号顺序创建的。刷新日志将导致服务器关闭当前日志文件和打开下一个顺序编号的新文件。

日志刷新操作对日志的失效或轮转处理工作中有着重要的意义，详见 12.5.7 节中的讨论。

日志刷新操作对于由各有关存储引擎管理的特殊用途日志没有影响。

12.5.1　出错日志

出错日志记载着 MySQL 服务器每次启动和关闭的时间以及诊断和出错信息。出错日志的信息量可以通过--log-warnings 选项来调控，该选项的可取值是从 0 到 2，而需要记载的信息量随这个值的大小而相应地增加或减少。

如果服务器把出错日志信息写入一个文件，FLUSH LOGS 语句将导致服务器把该文件重命名为带有-old 后缀，并用原来的文件名创建一个新文件来记载有关信息。

对于出错日志的其他特性，Unix 和 Windows 系统有着不同的处理方式，详见接下来的讨论。

1. Unix系统上的出错日志

在 Unix 系统上，mysqld 程序的默认行为是不创建出错日志，它会把诊断信息发送到控制台去。对于直接执行 mysqld 程序而启动的 MySQL 服务器，可以在命令行上或是在某个选项文件中的[mysqld]选项组里增加一个--log-error 选项，让它把出错信息写入一个文件而不是发送到控制台。

对于通过调用 mysql_safe 脚本而启动的 MySQL 服务器，出错日志是默认创建的，因为 mysql_safe 脚本在调用 mysqld 程序时会把服务器的输出重定向到出错日志。默认的出错日志文件名是 HOSTNAME.err，但你可以在命令行上或是在某个选项文件中的[mysqld_safe]或[mysqld]选项组里增加一个--log-error 选项，给出错日志另外起一个名字。(mysqld_safe 脚本会读取[mysqld]选项组并使用它在那里找到的--log-error 选项。)

mysql_safe 脚本和 mysqld 程序对--log-error 选项的处理有着细微的差异，所以在给出这个选项的时候要注意不要弄混了。

- 对于mysqld程序，在给出这个选项时可以省略文件名。此时，它将使用文件名HOSTNAME.err在数据目录里创建一个出错日志文件。对于mysql_safe脚本，必须在使用这个选项时给出一个文件名。
- 如果为--log-error选项给出了一个相对文件名，mysqld程序和mysql_safe脚本对它的解释是不同的。mysqld程序将把那个文件名解释为相对于数据目录而言，MySQL 5.1.11及更高版本里的mysql_safe脚本也是如此解释。但在MySQL 5.1.11之前的版本里，mysql_safe脚本将把那个文件名解释为相对于你调用该脚本时所在的子目录而言。因此，如果决定使用mysql_safe脚本来启动，但并不总是从同一个子目录里执行它的话（比如说，需要根据不同的情况在不同的地方手动执行），就应该使用一个绝对路径名来设定出错日志的名字以确保它

总是被创建在同样的地点。

- mysqld程序将自动添加一个扩展名.err，MySQL 5.1.11及更高版本里的mysql_safe脚本也会如此。但在MySQL 5.1.11之前的版本里，mysql_safe脚本将直接使用你给出的名字，不管它有没有扩展名。

如果出错日志文件已经存在，但用来运行服务器的那个登录账户对它没有写权限，MySQL 服务器的启动将会失败并且不会有任何输出消息被写入出错日志。这种问题在换用另外一个--user 值去启动 MySQL 服务器的时候最容易发生，所以最好使用同一个账户，详见 12.2.1 节的第 1 小节里的讨论。

从 MySQL 5.1.20 版开始，如果使用了 mysql_safe 脚本来启动服务器，你还可以把出错日志信息发送到 syslog 而不是发送到一个日志文件。（建议使用 MySQL 5.1.21 或更高版本，因为此前的版本所实现的这个功能有点儿小毛病。）为了把诊断信息发送到 syslog，请在执行 mysql_safe 脚本时用--syslog 选项取代--log-error 选项。在 syslog 日志里，来自 mysqld 程序和 mysql_safe 脚本的消息将分别带有一个 mysqld 或 mysql_safe 字样的标签（前缀）。如果你还使用了一个--syslog-tag=str 选项，这两个标签将变成 mysqld-str 或 mysql_safe-str。

如果执行 mysql.server 脚本来启动 MySQL 服务器，出错日志将总是创建好了的，这是因为 mysql.server 脚本调用了 mysql_safe 脚本的缘故。请注意，mysql.server 脚本不能识别在命令行上或是在它的[mysql.server]选项组里给出的与出错日志有关的任何选项。可以把必要的选项设置给 mysqld_safe 脚本或 mysqld 程序，具体做法见本节前面的内容。

2. Windows系统上的出错日志

在 Windows 系统上，MySQL 服务器的默认行为是把诊断信息写入数据目录中的 HOSTNAME.err 文件。如果你在以手动方式启动 MySQL 服务器时使用了--console 选项，它将把诊断信息写到控制台窗口并不再创建一个出错日志。（如果把 MySQL 服务器运行为一项 Windows 服务，--console 选项将没有任何效果，因为此时根本没有控制台可供写入。）

12.5.2 常规查询日志

这个日志记载着客户何时连接了服务器、客户向它发送的每一条语句以及各种各样的其他事件，如服务器的启动和关闭等。服务器按照它收到语句的先后顺序把它们写入这个日志，这个顺序与各语句执行完毕的先后顺序很可能不一样，尤其是简单的和复杂的语句相混杂的时候。

启用常规日志的方法是给出--log 选项。如果在给出这个选项时省略了文件名，服务器将默认使用数据目录里的 HOSTNAME.log 文件。

从 MySQL 5.1.6 版开始，可以选择把通用日志写到一个文件、一个数据库表或同时写到这两个地方。详见 12.5.6 节。

12.5.3 慢查询日志

慢查询日志记载着需要很长时间才能执行完毕的查询。

- 判断“很长时间”的标准是long_query_time系统变量的值（以秒为单位，默认值是10秒）。在MySQL 5.1.21之前的版本里，这个变量的最小值是1，默认值是10。从MySQL 5.1.21版开始，这个值可以有一个小数部分（以微秒为单位），而最小值变成了0。
- 从MySQL 5.1.21版开始，查询还必须至少验查min_examined_row_limit个数据行才有资格被记载到慢查询日志里。这个系统变量的默认值是0。

因为查询所需的时间直到它执行完毕时才知道，慢查询是在它们执行完毕后才记入日志的，不是在它们到达时记载。慢查询还会使得服务器递增它的 Slow-queries 状态变量。

慢查询日志是普通文本格式，所以可以用任何文件查看程序来阅读，也可以使用 mysqldumslow 实用工具程序对它的内容进行汇总。

慢查询日志可以帮助我们找出需要改写以改善其执行性能的查询。不过，在解读它的内容时，还应该把系统的正常工作负载考虑在内。“慢”是以实际时间（不是 CPU 时间）度量的，如果服务器的负载在某个时间段里突然增加，就可能会有许多查询被误判为是“慢”查询，而它们在系统负载处于正常情况时也许根本谈不上“慢”。

启用慢查询日志的方法是给出---log-slow-queries 选项。如果你在给出这个选项时省略了文件名，服务器将默认使用数据目录里的 HOSTNAME-slow.log 文件。

从 MySQL 5.1.6 版开始，可以选择把通用日志写到一个文件、一个数据库表或同时写到这两个地方。详见 12.5.6 节。如果是把慢查询信息写入一个数据表，查询执行时间的小数部分将被丢弃。

还有几个相关选项会影响到服务器将把哪些信息写入慢查询日志。--log-short-format 选项将导致服务器把较少的信息写入日志。--log-queries-not-using-indexes 选项将导致服务器把在执行时没有用到任何索引的查询也记载到慢查询日志里。--log-slow-admin-statements 选项将导致服务器把“慢”的系统管理语句如 ANALYZE TABLE 或者 ALTER TABLE 等也记载到慢查询日志。

12.5.4　二进制日志和二进制日志索引文件

MySQL 服务器使用二进制日志来记载数据修改“事件”，比如 INSERT、DELETE 或 UPDATE 等会导致数据发生变化的语句。它不会把 SELECT 操作记载到这个日志里。如下所示的 UPDATE 语句也不会出现在二进制日志里，因为没有任何数据因为它的执行而发生真正的变化：

```
UPDATE t SET i = i;
```

因为 MySQL 必须先执行一条语句才能知道它是否真的修改了数据，所以它把有关信息写入二进制日志的时间点，是语句结束执行之后而不是语句到达服务器之时。

二进制日志还包含着一些用于建立复制机制的信息，比如语句的执行时间戳。

与其他日志不同，二进制日志里的信息不是文本格式而是一种更有效率的二进制格式，这种二进制格式比文本占用的空间更少。这个日志使用二进制格式来记载信息的特点意味着人们无法直接阅读它的内容，但可以利用 mysqlbinlog 实用工具程序把二进制日志文件的内容显示为便于阅读的文本格式。

二进制日志可以用于数据库备份和恢复。另外，如果想把某个 MySQL 服务器设置为复制机制中的主服务器以便从服务器对之复制的话，就必须启用二进制日志。

MySQL 服务器按有关事件的执行顺序把它们写入二进制日志。换句话说，事件是按照它们执行结束时的先后顺序被写入二进制日志的，不是按照它们到达服务器时的先后顺序，这对复制机制的正确运转有着重要意义。对于构成事务的语句，服务器将把它们缓存起来直到该事务被成功提交，然后才会把在此期间发生的所有事件写入日志。如果事务被回滚了，它将不会被记载到二进制日志里，因为它没有导致数据库发生任何变化。

更准确的说法是：被回滚的事务通常不会被记载到二进制日志里。如果某个事务对非事务型数据表（比如 MyISAM 数据表）进行了修改，那些修改是无法回滚的。在这种情况下，即使事务被回滚了，它也会被记载到二进制日志里，这是为了确保复制机制中的从服务器能够正确地把非事务型数据表里

的变化重现出来。

启用二进制日志的方法是给出`--log-bin` 选项。如果在给出这个选项时省略了文件名，MySQL 服务器将使用 `HOSTNAME-bin` 作为基本文件名生成一组顺序编号的二进制日志文件：`HOSTNAME-bin.000001`、`HOSTNAME-bin.000002`，依此类推。否则，服务器将使用你给出的名字作为二进制日志的基本文件名（如果你给出的文件名包括一个扩展名，该扩展名将被忽略）。此后在你每次启动服务器或刷新日志的时候，以及当前二进制日志文件到达其最大长度的时候，服务器会按编号顺序生成下一个文件。二进制日志文件的最大长度由系统变量 `max_binlog_size` 的值确定。

如果启用了二进制日志功能，服务器还将创建一个配套的二进制日志索引文件，并在其中列出现有的二进制日志文件的名字。默认的索引文件名是二进制日志文件的基本名加上一个“.index”扩展名，但你可以利用`--log-bin-index` 选项给它另外起一个名字。如果你给出的文件名没有任何扩展名，MySQL 服务器将自动地给它加上一个“.index”扩展名。比如说，如果你给出的是`--log-bin-index=binlog`，二进制日志索引文件的名字将是 `binlog.index`。

如果你同时使用了`--log-short-format` 和`--log-bin` 选项，MySQL 将把较少的信息写入二进制日志。

在 MySQL 5.1 之前的版本里，二进制日志里记载的事件都是基于语句的；从 MySQL 5.1 版开始，可以选择使用基于语句的格式或基于数据行的格式来进行记载。比如说，在基于语句的日志里，一条 `UPDATE` 语句将被记载为一条 `UPDATE` 语句，但在基于数据行的日志里，一条 `UPFATE` 语句将被记载为这条语句对各有关数据行的改动。（请参阅 14.7.3 节。）用来设置日志格式的选项是`--binlog-format=format` 选项，允许使用的选项值是 `STATEMENT`、`ROW` 和 `MIXED`。从 MySQL 5.1.12 版开始，这个选项的默认值是 `MIXED`，其含义是以基于数据行的日志格式为主，只在确有必要时才使用基于语句的格式。

如果你正在使用二进制日志来实现复制机制，在你确认其内容已被复制到了所有从服务器并且今后也不再需要之前，千万不要删除它。我们将在 12.5.7 节的第 2 小节探讨如何确认。

二进制日志文件和系统备份

如果二进制日志（以及中继日志）因为硬盘故障而损坏或丢失，它们对数据库的崩溃恢复工作将起不到任何作用。因此，一定要养成定期对文件系统进行备份的好习惯。把这些日志和数据库分开写到不同的硬盘上也是个好主意，但这需要把它们的写入位置从服务器默认使用的数据目录改设为新硬盘。这并不难做到，只要通过日志选项另行指定一个新的日志写入位置就可以了。

12.5.5 中继日志和中继日志索引文件

复制机制中的从服务器把来自主服务器的数据修改信息（“事件”）在收到它们的同时写入它的中继日志。中继日志就像是一个临时收容所，数据修改信息在那里排队等待着从服务器执行它们。

在从服务器上，事件的接收和执行是由两个线程分别负责处理的。I/O 线程负责从主服务器接收事件并把它们写入中继日志，SQL 线程负责读取中继日志文件、执行事件以及在处理完每个文件后删除之。这两个线程在功能上互不干扰，这使它们可以彼此独立地运行。

中继日志和二进制日志有许多共同的特点。

- 从服务器按编号顺序创建中继日志文件。
- 有一个配套的索引文件用来列出当前正在使用的中继日志文件。

- 中继日志文件的格式与二进制日志文件相同，所以可以用mysqlbinlog实用工具程序显示它们的内容。

启用中继日志的方法是给出--relay-log 选项。如果在给出这个选项时省略了文件名，MySQL 服务器将使用 HOSTNAME-relay-bin 作为基本文件名生成一组顺序编号的中继日志文件：HOSTNAME-relay-bin.000001、HOSTNAME-relay-bin.000002，依此类推。否则，服务器将使用你给出的名字作为基本文件名（如果你给出的文件名包括一个扩展名，该扩展名将被忽略）。此后在你每次启动服务器或刷新日志的时候，以及当前中继日志文件到达其最大长度的时候，服务器会按编号顺序生成下一个文件。中继日志文件的最大长度由系统变量 max_relay_log_size 的值确定。

如果启用了中继日志，服务器还将创建一个配套的中继日志索引文件并在该文件里列出现有的中继日志文件的名字。默认的索引文件名是中继日志文件的基本名加上一个“.index”扩展名，但可以利用--relay-log-index 选项给它另外起一个名字。如果你给出的文件名没有任何扩展名，MySQL 服务器将自动地给它加上一个“.index”扩展名。比如说，如果你给出的是--relay-log-index=relay-log，中继日志索引文件的名字将是 relay-log.index。

12.5.6　日志数据表的使用

在 MySQL 5.1.6 之前的版本里，如果启用了常规查询日志或慢查询日志，服务器将把日志信息写到相应的日志文件里。从 MySQL 5.1.6 版开始，当启用这些日志时，可以选择把日志信息写到一个日志文件、mysql 数据库里的一个日志数据表或是同时写到这两处地方。（建议使用 MySQL 5.1.21 或更高版本，因为此前的版本所实现的这个功能有点儿小毛病。下面的讨论将假设你使用的是足够新的版本。）

在启动 MySQL 服务器的时候，可以使用--log-output=destinations 选项为日志信息挑选一个输出目的地，destinations 由该选项的一个或多个以逗号分隔的可取值构成：FILE（写入文件）、TABLE（写入数据表）或 NONE（两处地方都不写入）。如果给出了 NONE，它将覆盖任何其他的输出目的地。如果没有给出--log-output 选项或是在给出这个选项时省略了它的值，服务器将使用 FILE 作为该选项的默认值。

--log-output 选项只能用来为日志信息确定一个输出目的地，不能用来启用日志功能。启用常规查询日志或慢查询日志还是要使用--log 或--log-slow-queries 选项才行，就像当初还没有日志数据表这个概念时那样（请参阅 12.5.2 节和 12.5.3 节）。

日志信息的输出目的地还可以通过设置全局级 log_output 系统变量来改变。比如说，如果需要临时禁用日志功能，可以使用下面这条语句：

```
SET GLOBAL log_output='NONE';
```

要重新启用日志功能并使用文件和数据表作为日志信息的输出目的地，这么做：

```
SET GLOBAL log_output='FILE,TABLE';
```

如果启用了一个日志，服务器就会把启动消息写入相应的日志文件，但如果没有选择 FILE 作为日志信息输出目的地的话，服务器接下来将不会把任何查询记载到相应的日志文件里。

服务器使用 mysql 数据库里的 general_log 和 slow_log 数据表来实现其 TABLE 日志功能。（如果是从一个老版本升级到 MySQL 5.1.6 或更高版本的，千万记得运行 mysql_upgrade 脚本以确保这些数据表存在。）

全局级 general_log_file 和 slow_log_file 系统变量被设置为日志文件的名字。如果使用了 FILE 作为日志信息的输出目的地，改变这两个变量的值将导致服务器把相应的日志信息写入新名字所指定的文件。

日志数据表的内容只允许查看，不允许修改，除非服务器亲自动手。换句话说，只能把它们用在 SELECT 语句里，不能用在 INSERT、DELETE 或 UPDATE 语句里。（但可以使用 TRUNCATE TABLE 语句去清空一个日志数据表。）

12.5.7　日志管理

日志很重要，但启动日志时有一个危险，即有可能产生大量的信息，或许填满你的磁盘。当你有一个处理许多查询的服务器时，这就特别真实。为了保持最后几个日志在线可用，同时要防止日志文件无限增长，你可以使用日志文件过期失效技术。下列一些方法可用于保持日志可管理。

- **日志轮转**。这适用于具有固定文件名的日志文件，例如一般日志和慢查询日志文件。
- **基于工作期限的截止**。这方法去除比确定工作期限更早的日志文件。这适用于以编号序列建立的编号日志文件，例如二进制日志。但如果你使用了二进制日志用于复制，就不能使用这个技术。
- **有关复制的过期失效**。如果你的二进制日志文件用于复制，最好不要根据工作期限使该文件过期失效，而是应该在已知文件内容已复制至所有从服务器后才让它过期。所以这种形式的终止是根据二进制日志是否还在使用来决断的。

  复制从服务器按编号顺序创建中继日志文件，并在处理完它们之后自动删除它们。为了减少中继日志信息占用的硬盘空间，可以通过设置 max_relay_log_size 系统变量来降低中继日志文件的最大可允许长度。
- **日志数据表的截短和轮转**。如果选择把日志信息写入mysql数据库中的数据表，可以截短它们或者是重新命名并替换为空数据表。

为了确保缓冲区里的日志信息已被写入硬盘，日志轮转通常都要与日志刷新配合使用。对日志的刷新可以通过执行一条 mysqladmin flush-logs 命令或是发出一条 FLUSH LOGS 语句来完成。

在接下来的几节里，我们将对如何使用这几种失效方法进行描述。这里讨论的日志刷新示例脚本可以在 sampdb 发行版本的 admin 子目录里找到。

无论实际选用的是哪一种日志失效技术，都应该把日志文件的备份问题纳入考虑范围内。把数据恢复操作可能会用到的日志文件全都备份下来是个很好的主意，所以在把它们都备份下来之前不应该让它们失效。

1. 轮转一组名称固定不变的日志文件

MySQL 服务器把几种日志信息写至具有固定名的文件中，如一般查询日志文件和慢查询日志文件。为了终止固定名的日志，应使用日志轮转。你可以维护在线的最后几个日志，但以你选择的数量来限制，防止它们超出磁盘限量。

日志文件的轮转如下进行。假定一般查询日志文件名叫 qlog。在第一次轮转时，将 qlog 改名为 log.1，并且让服务器开始写一个新的日志文件。在第二次轮转时，把 qlog.1 改名为 qlog.2，qlog 改名为 qlog.1，并且让服务器开始写另一个新的 qlog 文件。用这种方法，每个文件通过名字 qlog.1，qlog.2 等轮转。当该文件达到某一轮转点时，就可让以前的文件覆盖它而使其过期。举例来说，如果你每日轮转该日志，并要保持日志一星期的工作量，则应该保持 qlog.1 至 qlog.7。在每次轮转时，

你应该让 qlog.7 过期失效，其方法是让 qlog.6 覆盖它，使变成新的 qlog.7。

日志轮转的频率及旧日志的数量取决于服务器的繁忙程度（工作服务器产生更多日志信息）和你愿意分配多少磁盘空间给旧日志。

在 Unix 上，你可以在服务器打开日志文件时重命名现有的日志文件。刷新日志会引起服务器关闭该文件并打开一个新文件，所以用原来的名字建立一个新的日志文件。下面的 shell 脚本 rotate_fixed_logs.sh 可以用于执行固定名日志文件的轮转：

```
#!/bin/sh
# rotate_fixed_logs.sh - rotate MySQL log file that has a fixed name

# Argument 1: log filename

if [ $# -ne 1 ]; then
  echo "Usage: $0 logname" 1>&2
  exit 1
fi

logfile=$1

mv $logfile.6 $logfile.7
mv $logfile.5 $logfile.6
mv $logfile.4 $logfile.5
mv $logfile.3 $logfile.4
mv $logfile.2 $logfile.3
mv $logfile.1 $logfile.2
mv $logfile $logfile.1
mysqladmin flush-logs
```

该脚本把日志文件名作为其参数。你可以指定文件的全路径名，或者把目录改成日志目录，指定该目录中的文件名。例如，为了轮转/usr/mysql/data 中的日志文件，执行以下脚本：

```
% rotate_fixed_logs.sh /usr/mysql/data/qlog
```

或者是下面的命令：

```
% cd /usr/mysql/data
% rotate_fixed_logs.sh qlog
```

说明　*在前几次执行日志轮转脚本的时候，因为日志文件的个数尚不足以让该脚本对之进行轮转，所以该脚本将报告说它无法找到用于轮转的所有文件。这是正常现象，不必担心。*

以运行服务器时的账号（本书中为 mysql 账号）登录，并运行该脚本，这可保证你有重命名日志文件的权利。注意，该脚本中的 mysqladmin 命令不包括连接参数，如-u 或-p。如果调用 mysqladmin 的连接参数存储在 mysql 账户主目录的.my.cnf 选项文件中，则不必在脚本程序中 mysqladmin 命令上指定它们。如果你没有使用选项文件，mysqladmin 命令需要知道如何使用有足够权限刷新日志的 MySQL 账号连接至服务器。为了做到这一点，要设置一个权限有限的账号，它不能做任何事，但可发出刷新命令。然后你可把该账号的口令放在脚本中，这有一个最小的风险，即 mysql 能访问该脚本。如果你想这么做，MySQL 账号只应该有 RELOAD 权限。例如，调用用户 flush 和赋予另一个 flushpass 口令，使用下列语句：

```
CREATE USER 'flush'@'localhost' IDENTIFIED BY 'flushpass';
GRANT RELOAD ON *.* TO 'flush'@'localhost';
```

建立该账号后，改变 rotate-fixed-logs.sh 脚本中的 mysqladmin 命令，如下所示：

```
mysqladmin -u flush -pflushpass flush-logs
```

要保护脚本不被其他登录账户读取，只能让 mysql 访问脚本。以 mysql 登录后执行以下命令：

```
% chmod go-rwx rotate_fixed_logs.sh
```

要了解如何使用 rotate_fixed_logs.sh 脚本定期轮转和刷该日志，参见稍后第 3 小节的内容。

在 Linux 下，最好使用 logrotate 实用工具程序来安装 MySQL 发行版本的 mysql-log-rotate 脚本，而不是使用 rotate-fixed-logs.sh 或写自己的脚本程序。然后查找 mysql-log-rotate（对于 RPM 发行版本是在/usr/share/mysql 中，对于二进制版本是在 MySQL 的 support-files 目录中，或者在 MySQL 源发行版本的 share/mysql 目录下）。

在 Windows 下，使用以下批脚本执行日志文件：

```
rotate_fixed_logs.bat:
@echo off
REM rotate_fixed_logs.bat - rotate MySQL log file that has a fixed name

REM Argument 1: log filename

if not "%1" == "" goto ROTATE
@echo Usage: rotate_fixed_logs logname
goto DONE

:ROTATE
set logfile=%1
erase %logfile%.7
rename %logfile%.6 %logfile%.7
rename %logfile%.5 %logfile%.6
rename %logfile%.4 %logfile%.5
rename %logfile%.3 %logfile%.4
rename %logfile%.2 %logfile%.3
rename %logfile%.1 %logfile%.2
rename %logfile% %logfile%.1
:DONE
```

rotate_fixed_logs.bat 的调用很像 rotate_fixed_logs.sh shell 脚本，它具有一个命名要轮转的日志文件的参数。若要轮转 C:\mysql\data 中的 qlog 日志文件，按如下所示执行脚本：

```
C:\> rotate_fixed_logs C:\mysql\data\qlog
```

或者执行如下所示的脚本：

```
C:\> cd \mysql\data
C:\> rotate_fixed_logs qlog
```

说明 在 MySQL 5.0.17/5.1.3 之前的版本里，在 Windows 系统上无法对已被服务器打开的常规查询日志或慢查询日志文件进行重命名，你会看到一条“文件正在使用”的出错消息。因此，在老版本的服务器上，必须确保日志文件没有被打开才能对它进行轮转。最稳妥的做法是：先停止 MySQL 服务器的运行，等执行完文件重命名操作后再重新启动服务器。

12

2. 二进制日志和中继日志文件的失效处理

名字固定的日志文件可以通过文件名轮转的办法来进行失效处理，就像刚才讨论的那样。对于二进制日志和中继日志等编号型日志，服务器将按编号顺序生成一系列日志文件，对它们进行失效处理需要另想办法。

对于二进制日志，有两种思路可供选择。

- 根据日志文件的“年龄”进行失效处理（“年龄”从最后一次修改时间算起）。如果没有把二进制日志用于复制机制的话，可以采用这个办法。
- 根据日志文件是否仍在使用进行失效处理。如果把二进制日志用在了复制机制里，这个办法更适用。

如果没有把二进制日志用于复制机制，对日志文件进行失效处理的最简单办法是设置 expire_logs_days 系统变量。如果这个变量的值 *n* 大于零，服务器将自动地对那些“年龄”超过 *n* 天的二进制日志文件进行失效处理并更新二进制日志索引文件。比如说，如果你想通过设置这个变量对已经有一个星期没修改过的二进制日志文件进行失效处理，把下面这些行添加到某个选项文件里即可：

```
[mysqld]
expire_logs_days=7
```

这样一来，MySQL 服务器就会在启动时和打开新日志文件时去检查是否需要对二进制日志文件进行失效处理。

如果把二进制日志用在了复制机制里，基于“年龄”的失效处理策略就不适用了。这时候，“年龄”已不能成为某个日志文件是否可以被删除的判断依据。假设某个从服务器需要关闭一段时间，在这段时间内主服务器无法向它发送某给定二进制日志文件的内容。万一这个从服务器在给定日志文件到达其失效期时仍没有开机，丢弃那个文件将使复制机制变得毫无意义。要想避免这样的问题，就应该把某个二进制文件的内容是否已被复制到了所有的从服务器作为判断是否应该对它进行失效处理的的依据。

这里有一个问题，由于 MySQL 复制的异步特性，主服务器自己并不知道有多少从服务器或哪些文件已传送给从服务器。主服务器不会清除还未发送至所连接的从服务器的二进制日志文件，但不保证给定的服务器已在某个时间连接好。这意味着你必须知道哪些服务器用作从服务器，然后连接至每一个，并发出 SHOW SLAVE STATUS 语句确定哪些主服务器的二进制日志文件目前正由从服务器来处理。（该文件名是 Mastre_Log_File 列中的数值。）可以删掉任何不再由某个从服务器使用的二进制日志。

为了解工作过程，假定你有下列情况。

- 本地服务器作为主服务器，它有两个从服务器S1和S2。
- 存在于主服务器的二进制日志文件分别具有从binlog.038到binlog.042的名字。
- SHOW SLAVE STATUS在S1上产生下列结果：

  ```
  mysql> SHOW SLAVE STATUS\G
  ...
  Master_Log_File: binlog.000041
  ...
  ```

  而在 S2 上产生下列结果：

```
mysql> SHOW SLAVE STATUS\G
...
Master_Log_File: binlog.000040
...
```

在这种情况下，从服务器仍然需要的最小编号二进制日志为 binlog.000040，所以任何更小编号的日志都可以去除。为此，连接至主服务器并发出下列语句：

```
mysql> PURGE MASTER LOGS TO 'binlog.000040';
```

这引起服务器删除所有编号比命名文件更小的二进制日志，按刚才所述的情况来讲，包括 binlog.000038 和 binlog.000039。

SHOW SLAVE STATUS 和 PURGE MASTER LOGS 语句都需要 SUPER 权限。

让中继日志文件失效不需要采取任何特别的行动。复制从服务器按编号顺序创建中继日志文件。它会在当前中继日志文件达到其最大可允许长度的时候（或是在刷新该日志的时候）自动创建一个新的中继日志文件，并在处理完老文件之后自动删除它们。不过，如果中继日志的最大可允许长度很大的话，当前文件就会变得很大。为了减少中继日志信息占用的硬盘空间，可以通过设置 max_relay_log_size 系统变量来降低中继日志文件的最大可允许长度。

3. 让日志失效工作自动完成

可以手动调用日志过期失效程序，如果你有一种调度该命令的方法来自动执行，就不必自己记住如何来运行它们。在 Unix 上，进行这项工作的一种方法是使用 cron 实用工具程序和设置一个确定失效调度的 crontab 文件。如果你不熟悉 cron，请查看相关的 Unix 手册，使用下列命令：

```
% man cron
% man crontab
```

你必须使用另一个命令来读取 crontab 文件格式：

```
% man 5 crontab
```

假设你想使用 rotate_fixed_logs.sh 脚本去轮转一个名为 qlog 的常规查询日志，该脚本安装在/home/mysql/bin 子目录里，日志文件存放在/var/mysql/data 子目录里。先以 mysql 用户的身份登录，然后使用如下所示的命令去编辑 mysql 用户的 crontab 文件：

```
% crontab -e
```

该命令允许你编辑现有 crontab 文件的副本（如果 cron 作业还没有设置，那可能是空的）。把各行添加到文件中，如下所示：

```
30 4 * * * /home/mysql/bin/rotate_fixed_logs.sh /var/mysql/data/qlog
```

这些项告知 cron，在每天早晨 4 点半运行这个脚本程序。你可以根据需要改变时间或调度，请查看 crontab 手册中的项格式。你或许想要为繁忙的服务器更频繁地轮转这些日志，因为繁忙的服务器比不忙的服务器有更多的日志信息。

如果要保证日志日常更新(例如,产生下一个带编号的二进制修改日志),你可以调度 mysqladmin flush_logs 命令，通过增加另一个 crontab 项来定期执行。可能需要列出至 mysqladmin 的完整路径名，保证 cron 能找到它。

在 MySQL 5.0.17/5.1.3 之前的版本里，在 Windows 系统上实现自动化日志文件轮转要麻烦得多，因为无法对已被服务器打开的常规查询日志或慢查询日志文件进行重命名（参见 12.5.7 节的第 1 小节）。这意味着如果想对任何一个当前日志文件进行轮转，就不得不先关闭、再重启服务器。这又进

一步意味着只能在没人使用 MySQL 服务器的时候才能进行这种轮转，可是往往很难保证 MySQL 服务器什么时候是没人使用的。

4. 日志数据表的失效或轮转处理

如果让 MySQL 服务器把常规查询日志或慢查询日志写入 mysql 数据库中的数据表，可以选择截短那些数据表或是采用某种方式对那些数据表进行轮转。

如果选择截短那些数据表，可以使用如下所示的语句：

```
USE mysql;
TRUNCATE TABLE general_log;
TRUNCATE TABLE slow_log;
```

如果选择对日志数据表进行轮转，需要先为它创建一份空白的副本，然后执行一次原子化重命名操作，用一条语句把当前数据表替换为它的空白副本：

```
USE mysql;
CREATE TABLE general_log_tmp LIKE general_log;
RENAME TABLE general_log TO general_log_old, general_log_tmp TO general_log;
CREATE TABLE slow_log_tmp LIKE slow_log;
RENAME TABLE slow_log TO slow_log_old, slow_log_tmp TO slow_log;
```

用于日志数据表的 RENAME TABLE 语句要求是 MySQL 5.1.13 或更高版本。

还可以利用事件调度程序（event scheduler）让日志数据表的轮转工作自动完成，这只需要创建一个如下所示的事件就行了。这个事件每天轮转日志一次。调整 ON SCHEDULE 子句就可以改变这个频率。

```
CREATE EVENT mysql.rotate_log_tables
ON SCHEDULE EVERY 1 DAY
DO BEGIN
  DROP TABLE IF EXISTS general_log_old, general_log_tmp;
  CREATE TABLE general_log_tmp LIKE general_log;
  RENAME TABLE
    general_log TO general_log_old,
    general_log_tmp TO general_log;
  DROP TABLE IF EXISTS slow_log_old, slow_log_tmp;
  CREATE TABLE slow_log_tmp LIKE slow_log;
  RENAME TABLE
    slow_log TO slow_log_old,
    slow_log_tmp TO slow_log;
END;
```

12.6　调整 MySQL 服务器

MySQL 服务器的某些系统变量（参数）会对它自己的运行情况产生影响。可以用 SHOW VARIABLES 语句来查看这些变量。如果这些变量的默认值不适合你的具体情况，你可以通过修改它们来为你的 MySQL 服务器配置更好的运行环境。在这些变量当中，有一些可以用来调整性能，比如那些用来控制内存缓冲区大小的变量。比如说，如果你的内存比较充裕，就应该加大服务器用于磁盘和索引操作的缓冲区大小。内存里容纳的信息越多，磁盘读写操作就会越少。反之，如果你的内存比较紧张，就应该减小缓冲区的大小，这往往会降低服务器的运行速度，但有助于改善系统的整体性能，因为这样能防止 MySQL 服务器占用太多系统资源，不利于其他进程。

其他的变量将影响 MySQL 服务器与客户之间的交互，比如那些用来控制 SQL 模式、默认存储引

擎和当前时区的变量。

服务器还有一些状态变量，它们提供的信息可以让我们及时掌握服务器的实际运转情况。可以用 SHOW STATUS 语句来查看这些变量。这些状态变量可以用来监视服务器，还可以用来检验通过修改系统变量而作出的配置调整的效果究竟如何。

接下来的几节将讨论 MySQL 服务器变量的设置和查看办法，以及一些普适性的 MySQL 服务器变量。12.7 节将讨论特定于存储引擎的参数调整问题。在《MySQL 参考手册》与优化调整有关的章节里也有很多有价值的信息。

12.6.1 查看和设置系统变量的值

绝大多数系统变量都可以在服务器启动时通过命令行上或选项文件里的选项进行设置，通用语法请见 F.1 节。在服务器正在运行时，可以用 SHOW VARIABLES 语句查看系统变量，还可以在服务器运行时动态地修改许多变量。这种动态修改能力让我们可以更好地控制服务器的操作，避免为了调整某些配置选项而不得不关停服务器。（比如说，当你试验不同的缓冲区大小对 MySQL 服务器的性能有什么影响时，就不必再像从前那样每改变一次设置值就得关停并重新启动一次服务器了。）运行时作的修改只能维持到服务器退出运行之时，因此，如果你为某个变量试出了一个效果比它的当前默认值更优的新值，就应该把新值写到选项文件里，让服务器以后总是用新值启动。

有几个系统变量会影响到服务器与客户的交互方式。客户可以通过改变这些变量来对服务器端的操作进行控制，让应用程序可以对它们需要的行为进行定制。

系统变量按其作用范围的大小分为两个级别：全局级和会话级。全局级变量将全面影响整个服务器的操作，会话级变量只影响服务器将如何对待一个给定的客户连接。如果某个变量同时存在于两个级别，MySQL 将在客户建立连接时（只能在此时）用全局级变量的值去初始化相应的会话级变量，但在客户连接建立起来之后，对全局级变量作出的修改将不会影响到相应的会话级变量的当前值。

有些系统变量有全局级和会话级两种形式，有些只有全局级形式，有些只有会话级形式。

- 系统变量sql_mode是同时存在于全局级和会话级的一个典型例子，它的作用是设置默认的SQL模式。SQL模式将对服务器在处理SQL语句时的一些具体做法产生影响。当一个客户连接到服务器时，它将获得它自己的会话级sql_mode变量，该变量的初始值将被设置为全局级sql_mode变量的值。任何一个客户都可以修改它自己的会话级变量值，这种修改只对服务器在该客户建立的连接上的行为产生影响，不会影响到服务器如何对待其他的客户。具备SUPER权限的客户可以修改全局级sql_mode变量，而服务器将使用新的全局级值对此后建立的客户连接的会话级变量进行初始化。
- 系统变量key_buffer_size是只存在于全局级的一个例子。它可以用来控制默认键缓存的大小，这个缓冲区的内容是MyISAM数据表的索引数据。这个键缓冲区是由全体客户共享的，所以没有必要让每个客户各有一个会话级的值。
- 有些变量只存在于会话级。autocommit变量就是其中之一。每个客户的autocommit模式在一开始时都是默认启用的，但可以在必要时禁用之。

附录 D 列出了所有的系统变量，并介绍了能否在服务器启动和运行时设置它们。接下来的几个小节将着重介绍如何查看和设置这些系统变量的语法。

1. 查看系统变量的值

系统变量的当前值可以用 SHOW VARIABLES 语句来查看：

```
mysql> SHOW VARIABLES;
+---------------------------------+-------------------+
| Variable_name                   | Value             |
+---------------------------------+-------------------+
| auto_increment_increment        | 1                 |
| auto_increment_offset           | 1                 |
| autocommit                      | ON                |
| automatic_sp_privileges         | ON                |
| back_log                        | 50                |
| basedir                         | /usr/local/mysql  |
| big_tables                      | OFF               |
...
```

我们还可以用一个 LIKE 子句让输出只列出那些名字与给定 SQL 模板相匹配的系统变量：

```
mysql> SHOW VARIABLES LIKE 'key%';
+--------------------------+---------+
| Variable_name            | Value   |
+--------------------------+---------+
| key_buffer_size          | 8388572 |
| key_cache_age_threshold  | 300     |
| key_cache_block_size     | 1024    |
| key_cache_division_limit | 100     |
+--------------------------+---------+
```

从 MySQL 5.0.2 版本开始，可以给 SHOW VARIABLES 语句增加一个 WHERE 子句来给出筛选数据行的条件。以下语句用于找到设置值小于 60 秒的 timeout：

```
mysql> SHOW VARIABLES
    -> WHERE Variable_name LIKE '%timeout%' AND Value < 60;
+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| connect_timeout            | 10    |
| innodb_lock_wait_timeout   | 50    |
| innodb_rollback_on_timeout | OFF   |
| net_read_timeout           | 30    |
| table_lock_wait_timeout    | 50    |
+----------------------------+-------+
```

SHOW VARIABLES 语句将优先显示系统变量在会话级的值，如果这个值不存在，则显示该变量在全局级的值。如果只想查看在全局级或会话级的值，必须在 SHOW VARIABLES 语句里明确地加上 GLOBAL 或 SESSION 关键字：

```
SHOW GLOBAL VARIABLES;
SHOW SESSION VARIABLES;
```

LOCAL 是 SESSION 的一个同义词。

mysqladmin variables 命令将显示 MySQL 服务器的全局级系统变量的当前值。

变量的值可以用@@GLOBAL.*var_name* 语法（全局级变量）或者@@SESSION.*var_name* 或@@LOCAL.*var_name* 语法（会话级变量）来查看。如果在变量名 *var_name* 的前面没有任何级别限定符，将优先显示它在会话级的值（如果该值存在）；否则，显示它在全局级的值。

@@_语法是通用的，它们还可以用在 SET、SELECT 或其他 SQL 语句里，如下所示：

```
SELECT 'Default storage engine:', @@storage_engine;
```

绝大多数只存在于会话级的变量根本不会出现在 SHOW VARIABLES 语句的输出里，但我们可以通过它们的名字去访问它们的值：

```
SELECT @@autocommit, @@warning_count;
```

从 MySQL 5.1.12 版本开始，还可以通过查询 INFORMATION_SCHEMA 数据库里的 GLOBAL_VARIABLES 和 SESSION_VARIABLES 数据表来获得系统变量的信息。

2. 在启动服务器时设置系统变量的值

有许多全局级系统变量都可以在服务器启动时动态设置。可以用来进行这种设置的语法有两种。

- 可以把一个变量名视为一个选项名直接设置它的值。比如说，MySQL服务器能够同时应付的客户连接的最大数量由max_connections变量控制，如果想把这个变量设置为200，可以在mysqld命令行里增加一个如下所示的选项：

  ```
  % mysqld --max_connections=200
  ```

  也可以按如下所示的语法在某个选项文件设置这个变量：

  ```
  [mysqld]
  max_connections=200
  ```

  这种“把变量视为选项”的语法还允许把变量名中的下划线（_）写成一个连字符（-），让服务器变量看上去与真正的选项没有什么不同。比如下面这个命令行“选项”：

  ```
  % mysqld --max-connections=200
  ```

  在选项文件里也可以这样设置：

  ```
  [mysqld]
  max-connections=200
  ```

- 还有一种设置变量的办法，在启动服务器时使用--set-variable或-O选项。这种语法已经有些过时了，但仍可以使用。在命令行上，要像下面这样设置变量：

  ```
  % mysqld --set-variable=max_connections=200
  % mysqld -O max_connections=200
  ```

  在选项文件里，只允许使用选项的长格式：

  ```
  [mysqld]
  set-variable=max_connections=200
  ```

与在命令行上设置系统变量的做法相比，在选项文件里设置系统变量往往更简明易用，因为你将用不着记住在每次启动服务器时都需要设置哪些变量。

在设置那些代表缓冲区大小或长度的变量时，如果设置值不带任何后缀，则以字节为单位；如果带有“K”、“M”或“G”等字母作为后缀，则分别以 KB、MB 或 GB 为单位。这些后缀不区分字母的大小写情况，所以也完全可以把它们写成“k”、“m”或“g”。

有些系统变量是无法使用一个启动选项去直接设置的，但通常会有一个相关的选项可以让你绕开这一限制。比如说，我们不能在启动服务器时直接设置 storage_engine 变量，但--default-storage-engine 选项可以让我们达到目的。附录 D 说明了哪些系统变量可以直接设置。对于那些不能直接设置的系统变量，该附录列出了与之相关的设置选项（如果存在这样一个选项的话）。

3. 在运行时设置系统变量的值

用来在运行时设置系统变量的语法取决于你打算设置一个全局级变量还是一个会话级变量。给定一个全局级变量 *var_name*，下面任何一种 SET 语句都能对它进行设置：

```
SET GLOBAL var_name = value;
SET @@GLOBAL.var_name = value;
```

类似地，针对会话级变量的 SET 语句也有两种格式，如下所示：

```
SET SESSION var_name = value;
SET @@SESSION.var_name = value;
```

LOCAL 是 SESSION 的一个同义词。

不带级别限定符的 SET 语句修改的是会话级变量，如下所示：

```
SET var_name = value;
SET @@var_name = value;
```

可以用一条 SET 语句去设置多个变量，但要用逗号把各变量赋值语句隔开，如下所示：

```
SET SESSION sql_warnings = 0, GLOBAL storage_engine = InnoDB;
```

在一条用来设置多个变量的 SET 语句里，明确给出的 GLOBAL 或 SESSION 级别限定符将作用于随后的所有变量设置，直到遇见下一个级别限定符。下面的语句将设置全局级变量 *v1* 和 *v2* 以及会话级变量 v3 和 v4：

```
SET GLOBAL v1 = val1, v2 = val2, SESSION v3 = val3, v4 = val4;
```

必须具备 SUPER 权限才能对全局级变量进行设置，新值的效果将持续到该变量被再次修改或 MySQL 服务器退出执行。设置会话级变量不需要特殊的权限，新值的效果将持续到该变量被再次修改或当前连接断开。

和在启动服务器时设置的变量不同，在运行时设置的变量不允许使用后缀字母“K”、“M”或“G”，但可以使用表达式，而且表达式还可以引用其他变量的值。下面的语句将把全局级系统变量 read_buffer_size 的值设置为 2MB，把会话级同名变量的值设置为 4MB：

```
SET GLOBAL read_buffer_size = 2*1024*1024;
SET SESSION read_buffer_size = 2*@@GLOBAL.read_buffer_size;
```

有许多系统变量可以被设置为特殊的 DEFAULT 值。对于那些支持这种语法的变量，把 DEFAULT 赋给一个全局级变量，将把它设置为编译阶段设定的默认值（注意，这个默认值与你在启动服务器时用启动选项设置的值不一样）。把 DEFAULT 赋给一个会话级变量，将把它设置相应的全局级变量的当前值。

MySQL 还支持结构化的系统变量，后者由一组彼此相关的系统变量构成，组织和读写这些变量需要使用对结构变量进行访问的语法。现有的结构化系统变量都是用来配置 MyISAM 键缓冲区的，我们将在 12.7.2 节介绍它们的语法。

12.6.2　通用型系统变量

有些系统变量在优化 MySQL 服务器的整体性能时很有用，下面是它们当中最常用的几个。

- delayed_queue_size

  在被实际插入到各有关数据表（对于那些支持DELAYED插入的存储引擎）里去之前，来自

INSERT DELAYED语句的数据行将在每个队列里等待MySQL来处理它们。delayed_query_size就是这个队列所能容纳的数据行的最大个数。当这个队列满时，后续的INSERT DELAYED语句将被阻塞，直到这个队列里有容纳它们的空间为止。如果有很多客户在发出INSERT DELAYED语句以避免受阻塞，但你发现这些语句有阻塞的迹象，加大这个变量的值将使更多的INSERT DELAYED语句更快地得到处理。（对INSERT DELAYED语句的详细介绍见5.5节。）

- max_allowed_packet

  MySQL服务器在与客户（程序）之间进行通信时使用的缓冲区的最大长度。这个变量的最大可取值是1GB。MySQL服务器使用的这种通信缓冲区的默认最大长度是1 MB，有些客户还可以有它们自己的max_allowed_packet变量。如果你的客户经常批量传输一些非常长的语句（比如说，包含着超长BLOB或TEXT值的语句），你可能需要在服务器端和客户端同时加大这个变量的值。比如说，把下面这些语句添加到MySQL服务器的选项文件里将使它以64 MB作为数据包的长度上限：

  ```
  [mysqld]
  max_allowed_packet=64M
  ```

  如果只是偶尔需要使用 64MB 作为数据包的长度上限来调用 mysql 或 mysqldum 客户程序，可以这么做：

  ```
  % mysql --max_allowed_packet=64M ...other options...
  % mysqldump --max_allowed_packet=64M ...other options...
  ```

  如果总是要用到这些设置，就应该把下面这些代码添加到你的选项文件里：

  ```
  [mysql]
  max_allowed_packet=64M

  [mysqldump]
  max_allowed_packet=64M
  ```

- max_connections

  这是MySQL服务器允许同时处于打开状态的客户连接的最大个数。如果你的MySQL服务器很繁忙，你可能需要加大这个值。比如说，假设你的MySQL服务器被一个繁忙的Web服务器用来处理大量的由DBI或PHP脚本生成的数据库查询命令，而你把这个变量设置得很小，你网站的访问者们就会发现他们的请求经常被拒绝。

- table_cache（从MySQL 5.1.3版本开始，这个变量改名为table_open_cache）

  这是数据表缓存的尺寸。加大这个值，将使mysqld能够同时打开更多的数据表，从而减少文件打开/关闭操作的次数。

如果你加大了 max_connections 或 table_cache/table_open_cache 变量的值，MySQL 服务器就会占用更多的文件描述符，可如果你的操作系统对每个进程所能占用的文件描述符的个数有限制，你这样做就会引起一些问题。如果真的遇到这种问题，你就需要加大操作系统对每个进程所能占用的文件描述符的个数限制或者另想办法。增加文件描述符个数的具体做法取决于具体的系统环境。你还可以试试为 MySQL 服务器程序 mysqld 设置 open_files_limit 变量的办法。如果用那些办法设置的打开文件上限还不够大，你恐怕需要重新配置系统才能允许占用更多的文件描述符。有些系统比较容易配置，只要编辑一个系统描述文件再重新开机就能达到目的；另外一些系统则要复杂一些，

可能需要编辑一个内核描述文件并重新构建系统内核才能奏效。具体做法请参考有关的系统文档。

还有一种办法能让你解决文件描述符的个数限制：把一个 MySQL 数据目录分割成多个并运行多个 MySQL 服务器。从效果上讲，这等于是把每个进程可用的文件描述符个数与你运行的 MySQL 服务器个数相乘。但这个办法做起来比较复杂，并且很可能会带来其他一些问题。比如说，因为某个给定的 MySQL 服务器只能访问它自己的数据目录，所以你将不能通过它去访问其他数据目录里的数据库；再比如说，如果你想让你的用户能够访问多个 MySQL 服务器，就必须把他们的权限依次复制到各有关 MySQL 服务器的权限表里去才行。

另一种思路是减少 MySQL 服务器对文件描述符的需求量：为主 MySQL 服务器建立一个或多个从服务器，数据更新操作仍全部留在主服务器上完成，数据查询操作则分散到那些从服务器上进行。这将减轻主服务器上的客户负载，减少它对文件描述符的需求量。

有些变量是用来把系统资源分配给每一个 MySQL 客户程序的——如果你的系统经常有大量的 MySQL 客户程序在同时运行，加大这些变量的值会导致 MySQL 服务器的资源占用总量急剧升高。比如说，为改善 MySQL 访问性能，MySQL 管理员可能会加大 read_buffer_size 和 sort_buffer_size 变量的值，前者用来设置读操作所使用的缓冲区的尺寸，后者用来设置排序操作所使用的缓冲区的尺寸。但是，因为 MySQL 会给每个客户连接都分配这两个缓冲区，所以如果把这两个变量的值设置得过大的话，性能反而会因为急剧增加的系统资源消耗量而受到损害。

因此，在改变这种“每个连接都会有”的缓冲区的尺寸时一定要谨慎从事。应该逐步加大有关变量的值并在修改生效后立刻测试其效果，千万不要一下子把它们设置得太大。只有这样，才能在不严重损害系统性能的前提下给有关变量设置一个最适当的值。还要注意的是，一定要按实际情况去进行测试。这类缓冲区都是“按需分配”而不是客户连接刚一建立就分配的。比如说，如果某个客户在进行查询时没有要求进行排序操作，MySQL 就不会为它分配供排序使用的缓存区。再比如说，没有使用索引的联结操作需要用到一个缓冲区，这个缓冲区的长度由 join_buffer_size 变量控制。如果客户没有进行任何联结操作，就不会为它分配任何联结缓冲区。（反过来讲，如果某个客户打算执行一个涉及多个数据表的复杂的联结操作，它可能需要同时用到多个联结缓冲区。）因此，用来进行测试的客户（程序）、进行测试的时间和测试时使用的语句必须与实际情况保持一致才能让你们看到对有关变量的修改在服务器的内存需求方面产生的真实效果。

12.6.3　查看状态变量的值

MySQL 服务器提供的状态变量使我们可以及时掌握它的实际运行状况。可以使用 SHOW STATUS 语句来查看这些变量：

```
mysql> SHOW STATUS;
+-----------------------------------+----------+
| Variable_name                     | Value    |
+-----------------------------------+----------+
| Aborted_clients                   | 0        |
| Aborted_connects                  | 1        |
| Binlog_cache_disk_use             | 0        |
| Binlog_cache_use                  | 3        |
| Bytes_received                    | 125      |
| Bytes_sent                        | 151      |
| Com_admin_commands                | 0        |
...
```

从 MySQL 5.0.2 版本开始，状态变量也有了全局级和会话级两种值（就像系统变量那样），SHOW STATUS 语句也开始支持 GLOBAL 和 SESSION 限定符：

```
SHOW GLOBAL STATUS;
SHOW SESSION STATUS;
```

GLOBAL 变量与整个服务器（全体连接的总和）的状态有关，SESSION 变量只与当前连接的状态有关。默认的级别限定符是 SESSION。LOCAL 是 SESSION 的一个同义词。

如果想让一条包含着级别限定符的 SHOW STATUS 语句可以用在 MySQL 5.0.2 之前的版本里，可以把级别限定符写成一条版本注释。如下所示：

```
SHOW /*!50002 GLOBAL */ STATUS;
```

如果某个变量只有一个全局级的值，使用 GLOBAL 或 SESSION 限定符查看到的值将是一样的。附录 D 对每一个状态变量都有哪个级别的值进行了说明。

类似于 SHOW VARIABLES 语句，在 SHOW STATUS 语句里也可以用一个 LIKE 子句让它只列出那些名字与给定 SQL 模板相匹配的状态变量。比如说，下面这条语句将只列出与 MyISAM 键缓存有关的状态变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Key%';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| Key_blocks_not_flushed | 0     |
| Key_blocks_unused      | 7247  |
| Key_blocks_used        | 13    |
| Key_read_requests      | 41    |
| Key_reads              | 14    |
| Key_write_requests     | 40    |
| Key_writes             | 2     |
+------------------------+-------+
```

从 MySQL 5.0.2 版本开始，可以给 SHOW STATUS 语句增加一个 WHERE 子句来给出筛选数据行的条件。

从 MySQL 5.1.12 版本开始，还可以通过查询 INFORMATION_SCHEMA 数据库里的 GLOBAL_STATUS 和 SESSION_STATUS 数据表的办法来获得关于状态变量的信息。如下所示：

```
SELECT * FROM INFORMATION_SCHEMA.GLOBAL_STATUS
WHERE VARIABLE_NAME LIKE 'Qcache%' OR VARIABLE_NAME LIKE 'Key%';
```

状态变量只能由服务器设置，它们对用户来说是只读的，我们不能像对待系统变量那样使用 SET 语句来设置或修改它们。

12.7 存储引擎的配置

MySQL 服务器支持多种存储引擎。数据库管理员可以非常灵活地决定把哪些存储引擎提供给用户使用。关于存储引擎的概括性介绍见第 2 章。下面讨论如何通过配置手段为 MySQL 服务器挑选存储引擎，并提供针对 MyISAM、InnoDB 和 Falcon 存储引擎的配置信息。

12.7.1 为MySQL服务器挑选存储引擎

MySQL 服务器灵活控制对存储引擎的挑选：

- 在从源代码开始建立MySQL的时候，可以选择建立哪些存储引擎。
- 在启动任何一个MySQL服务器的时候，不管它是不是你本人编译出来的，你都可以有选择地激活或者禁用一个或多个存储引擎。把用不着的存储引擎禁用掉可以减少MySQL服务器的内存占用量。（特别提醒：如果禁用了某个存储引擎，就不能访问以前用这个存储引擎创建的数据表了。）
- 在MySQL服务器正在运行的时候，用户可以实时查询都有哪些存储引擎可供选用以及默认的存储引擎是哪一个。

表 12-7 列出了从源代码开始建立 MySQL 时供 `configure` 脚本用来包括或排除各存储引擎的编译配置选项。有些存储引擎即使已经被编译出来，但如果你不想使用它们的话，还可以在运行 MySQL 服务器程序（即 `mysqld`）时用一个启动选项把它们禁用掉。Falcon 存储引擎只能在 MySQL 6.0 或更高的版本里使用。`--skip-archive` 选项只能在 MySQL 5.1 或更高的版本里使用。

表 12-7　存储引擎的编译配置选项和运行时选项

存储引擎	编译配置选项	启动选项
ARCHIVE	`--with-archive-storage-engine` `--without-archive-storage-engine`	`--skip-archive`
BLACKHOLE	`--with-blackhole-storage-engine` `--without-blackhole-storage-engine`	
CSV	`--with-csv-storage-engine` `--without-csv-storage-engine`	
EXAMPLE	`--with-example-storage-engine` `--without-example-storage-engine`	
Falcon	`--with-falcon-storage-engine` `--without-falcon-storage-engine`	`--skip-falcon`
FEDERATED	`--with-federated-storage-engine` `--without-federated-storage-engine`	
InnoDB	`--with-innodb` `--without-innodb`	`--skip-innodb`
MyISAM	总是建立	总是启用
MEMORY	总是建立	总是启用
MERGE	总是建立	总是启用

MyISAM 存储引擎总是可用的。它既不能在编译阶段被排除在外，也不能在 MySQL 服务器启动时被禁用。（MySQL 数据库系统的权限表都是些 MyISAM 数据表，必须在 MyISAM 存储引擎可用的情况下才能读取它们。）MERGE 和 MEMORY 存储引擎也总是可用的。

因为人们对 MySQL 的开发和完善一直没有停止过，所以在比较新的 MySQL 源代码发行版本里可能会有上面这个表格没有收录的其他存储引擎可供选用。在 MySQL 源代码发行版本的顶层目录里，可以用下面这条命令把可编译的存储引擎查出来：

```
% ./configure --help
```

如果想知道哪些选项可以在启动服务器时用来启用或禁用某个特定的存储引擎，可以使用如下所示的命令：

```
% mysqld --verbose --help
```

MySQL 服务器在启动时会指定一个默认的存储引擎，那些在创建时没有明确给出 `ENGINE =`

engine_name 选项的数据表都将由这个存储引擎负责管理。在编译阶段指定的默认存储引擎是MyISAM，但我们可以在服务器启动时和运行时改用另一个默认的存储引擎。

可以在启动 MySQL 服务器时用--default-storge-engine=engine_name 选项让它改用另一个默认的存储引擎。比如说，如果想把 InnoDB 设置为默认的存储引擎，在服务器的选项文件里加上如下所示的代码即可：

```
[mysqld]
default-storage-engine = innodb
```

在 MySQL 服务器启动之后，可以用如下所示的语句之一来改用另一个默认的存储引擎：

```
SET GLOBAL storage_engine = engine_name;
SET SESSION storage_engine = engine_name;
```

第一条语句必须具备 SUPER 权限才能使用，它将改变此后建立的所有连接的默认存储引擎。执行第二条语句不需要具备任何特殊权限，它只影响当前客户会话，任何客户都可以使用这条语句来改变它自己的默认存储引擎。

下面这条语句可以用来查看全局级和会话级默认存储引擎分别是哪一个：

```
SELECT @@GLOBAL.storage_engine, @@SESSION.storage_engine;
```

如果想知道都有哪些存储引擎可供选用，可以使用 SHOW ENGINES 语句或者查询 INFORMATION_SCHEMA 数据库里的 ENGINES 数据表，参见 2.6.1 节中的第 1 小节。

12.7.2 配置MyISAM存储引擎

MyISAM 数据表的数据文件和索引文件是分别存放的，对它们的处理也各不相同。

- 对于从MyISAM数据表读出或被写入MyISAM数据表的数据，MySQL服务器将依赖于操作系统的文件系统所提供的缓存机制来缓存它们。
- 对于MyISAM数据表的索引数据，MyISAM存储引擎将把它们放在它自己的键缓存区里进行管理，这个缓存区是MyISAM存储引擎最重要的可配置资源之一。键缓存区的基本用途之一是帮助完成基于索引的数据检索和排序操作，它的另一个主要用途是帮助完成索引的创建和修改操作。

本节将对键缓存区的基本操作以及用来对它进行配置的系统变量进行介绍。

MyISAM 存储引擎的键缓存区的工作情况如下所示。

- 键缓存区的初始状态全部为空。
- 在服务器执行某个语句时，若需要用到某个数据表的索引值，它将检查那些值是否已被读取到键缓存区里。如果是，它将从内存里读出索引值并使用。否则，它将读取该数据表的索引文件，把索引值从硬盘读到键缓存区中的某个缓冲块里。
- 如果键缓存区已经满了、但还需要读入新的索引值，MySLQ 服务器将必须丢弃某个缓冲块里现有的索引值。在默认的情况下，服务器将根据 LRU（Least Recently Used，最近最少使用）算法来决定它应该丢弃哪些索引值。也就是说，服务器将选择最长时间没有被使用的缓冲块。键缓存区里的缓冲块按照它们最近一次被访问的时间构成一个链表，所以服务器只需选中位于链表尾部的那个缓冲块就可以达到目的。
- 如果被选中的缓冲块自被读入内存后从没被修改过，MySQL服务器将直接读入的新的索引值，

覆盖该缓冲块里的老数据。否则，MySQL服务器将先把该缓冲块里的老数据写入相应的索引文件再读入新索引值覆盖它们。

在键缓存区里没能找到的所需要的索引值被称为“脱靶值”，必须从硬盘读取；能找到的索引值被称为“命中值”。建立键缓存区的目的是为了减少对硬盘的访问次数（也就是降低“脱靶率”）。因为内存的访问速度要远远大于硬盘的访问速度，所以键缓存区可以显著改善数据库系统的性能。

某个缓冲块所包含的索引值被使用得越频繁，它在键缓存区里的存留时间就越长。大的键缓存区不仅可以提高命中率，还可以降低需要丢弃一些缓冲块的老内容以容纳新索引值的几率，进而减少在对索引进行处理时需要去访问硬盘的次数。如果内存比较充裕而 MySQL 服务器的键缓存区比较小，加大键缓存区的尺寸往往是最简单和最有效的配置调整措施。

键缓存区的尺寸可以通过设置 key_buffer_size 系统变量的办法来调整。它的默认值是 8 MB，但如果内存充足的话，最多可以把它加大到 4 GB。比如说，如果想把键缓存区设置为 512 MB，把如下所示的代码添加到一个选项文件里就可以了：

```
[mysqld]
key_buffer_size = 512M
```

但要特别注意的是，千万不要让键缓存区的长度超过系统可用内存的总量。那会导致键缓存区本身被交换出内存，与利用缓存区让信息驻留在内存的目的是背道而驰的。别忘了，MySQL 服务器还需要为供其他存储引擎使用缓存区而分配内存，而服务器主机还需要为运行在该主机上的其他进程分配内存。

前面的讨论很容易让人们认为只有一个键缓存区。但事实并非如此，MySQL 实际支持的键缓存区往往会有好几个，这可以为缓存操作提供更细致的控制：

- 用户可以只使用默认的键缓存区，也可以创建多个键缓存区；
- 用户可以对缓存区的总长度、缓冲块的单位长度和缓冲块的丢弃算法进行调控；
- 用户可以把一个或多个数据表关联到某个特定的键缓存区；
- 用户可以把一个或多个数据表的索引预先加载到某个特定的键缓存区。

创建多个键缓存区的目的是为了降低键缓存区本身的占用冲突。如果某个或某几个数据表上的访问量很大，为它们分配一个专用的键缓存区将使它们不再与其他的数据表竞用默认键缓存区。

每个键缓存区都与一组相应的系统变量相关联。因为那些变量是彼此相关的，所以它们被组织为一个形成结构化变量的元素。结构化变量是对简单变量的一种扩展，对它们的访问必须通过键缓存区名加变量名的语法来进行，如下所示：

cache_name.var_name

每个键缓存结构变量由以下元素构成。

- key_buffer_size。键缓存的总长度，以字节为单位。
- key_cache_block_size。键缓存块的单位长度，以字节为单位。在默认的情况下，这个长度值是1024个字节。
- key_cacher_limit。它控制着缓存块重用算法。如果把这个变量的默认值设置为100，MySQL将使用“最近最少使用”算法来确定应该重新使用哪一个缓存块。如果把这个变量的值设置为小于100，MySQL将使用“中点插入”（midpoint insertion）算法把这个缓存区切割成“暖链”（warm chain）和“热链”（hot chain）两个部分。key_cacher_limit值将被视为暖缓存链键缓存百分比，值在1和100之间。

在使用中点插入算法来划分暖链和热链的时候，访问最为频繁的缓存块将尽可能地被留在热链里。随着某个缓存块的访问量的增加或减少，MySQL 服务器将把它挪到热链或暖链里去，而对缓存块的重新使用和覆盖总是发生在位于暖链里的缓存块身上。

- key_cache_age_threshold。如果键缓存区的热链里的某个缓存块在这个变量所设定的时间里没有被访问过，MySQL服务器就会把它调整到暖链里去。这个变量的值越大，缓存块在热链里停留的时间就越长。这个变量的默认值是300，最小值是100。

MySQL 的默认键缓存只有一个，它的名字是“default”。如果你在引用一个键缓存成员变量时没有给出一个缓存的名字，MySQL 将认为你引用的是默认键缓存。也就是说，key_buffer_size 和 default.key_buffer_size 代表的是同一个值。键缓存的名字必须是一个合法的 MySQL 标识符，它们不区分大小写。它们可以像任何其他的标识符那样括在引号里引用（具体规则见 2.2 节）。

要想创建一个新的键缓存，只要把一个值赋值给相关变量的成员之一即可。比如说，如果你想在启动 MySQL 服务器时创建一个名为 my_cache 的键缓存并把它的长度设置为 24MB,把下面这些语句添加到 MySQL 服务器的选项文件里就行了：

```
[mysqld]
my_cache.key_buffer_size = 24M
```

如果想在 MySQL 服务器运行时创建一个新的键缓存，请使用如下所示的语句：

```
SET GLOBAL my_cache.key_buffer_size = 24*1024*1024;
```

在这里，"GLOBAL"关键字必不可少，这是因为键缓存都是全局性的。访问各成员变量的值不需要具备任何特殊的权限，但要想设置它们就必须具备 SUPER 权限才可以。

在创建出一个键缓存之后，就可以用 CACHE INDEX 语句把 MyISAM 数据表分配给它了。这条语句需要提供一个键缓存的名字和一个或多个数据表的名字。下面的示例语句将把 sampdb 数据库里的 member 和 president 数据表分配给名为 my_cache 的键缓存：

```
CACHE INDEX member, president IN my_cache;
```

接下来，如果你愿意，你还可以用 LOAD INDEX INTO CACHE 语句把数据表的索引提前加载到指定的键缓存里去：

```
LOAD INDEX INTO CACHE member, president
```

把索引值提前加载到键缓存并不是必要的步骤，但如果真的这么做了，MySQL 服务器将按顺序读入索引数据块。这比等到有必要时才读入它们的效率要高一些。

必须具备某给定数据表的 INDEX 权限，才能使用 CACHE INDEX 为该数据表分配键缓存，才能使用 LOAD INDEX INTO CACHE 语句把该数据表的索引提前加载到指定的键缓存。

如果想删除某个现有的键缓存，把它的长度设置为零即可。此前已经分配给这个缓存的所有数据表都将被重新分配给默认的键缓存。如果把默认键字缓存区的长度设置为零，已经分配给它的数据表索引将由文件系统的缓存机制负责处理，就像对待 MyISAM 数据文件那样。

键缓存的分配效果只能持续到 MySQL 服务器被关停。如果想在 MySQL 服务器每次启动后都能获得同样的键缓存分配效果，最好的办法是把相应的 CACHE INDEX 和 LOAD CACHE INTO CACHE 语句放到一个文件里并在启动 MySQL 服务器时用一个--init-file 选项来导入那个文件。

12.7.3　配置InnoDB存储引擎

InnoDB 存储引擎使用一个共享的表空间来存储所有 InnoDB 数据表的内容和它的数据字典（data dictionary）。我们也可以通过配置 InnoDB 存储引擎的办法让它为每个 InnoDB 数据表分别使用一个表空间。InnoDB 存储引擎有它自己的日志文件和内存缓冲机制。

1. 配置InnoDB表空间

在默认的情况下，InnoDB 存储引擎不像 MyISAM 等其他存储引擎那样会为每个数据表分别创建一些文件，它会把所有的 InnoDB 数据表都存放在同一个共享的表空间里，表空间是一个在逻辑上为整体的存储块，InnoDB 存储引擎把它当做一个巨大的数据结构来对待。（从某种意义上讲，InnoDB 表空间就像是一个虚拟的文件系统。）对存储在共享表空间里的 InnoDB 数据表来说，与该数据表相关联的文件只有它的`.frm` 格式文件，这个`.frm` 文件存放在该 InnoDB 数据表所属的数据库的数据库子目录里。

InnoDB 存储引擎也可以配置成把每个数据表用它自己的表空间文件来表示。换句话说，数据表可以用它们各自的表空间来创建。如果想让 InnoDB 存储引擎为每个数据表分别创建一个表空间，在启动 MySQL 服务器时使用`--innodb-file-per-table` 选项即可。但即便如此，那个共享的表空间也必不可少，因为 InnoDB 存储引擎还需要把它的数据字典保存在其中，虽然它的尺寸用不着那么大了。

- InnoDB共享表空间的配置参数

InnoDB 共享表空间在逻辑上被视为一个存储区域，但实际上却是由一个或者多个磁盘文件组成的。各组成文件既可以是一个常规文件，也可以是一个未经格式化的原始硬盘分区。在这一节里，我们将对用来创建和管理 InnoDB 共享表空间的配置选项进行介绍。你们当然可以在 MySQL 服务器的启动命令行上设定这些选项，但这种做法在实践中很少见。为保证自己在每次启动 MySQL 服务器时使用的都是同样的 InnoDB 配置，应该把 InnoDB 表空间的配置放到某个选项文件里的某个适当的服务器选项组（比如`[mysqld]`或`[server]`选项组）里去。下面是最重要的两个配置选项。

- ❑ `innodb_data_home_dir`。用来为构成InnoDB表空间的所有组成文件指定一个父目录（即所谓的“InnoDB主目录”）。如果你没有给出这个选项，它的默认值将是相应的MySQL数据目录。
- ❑ `innodb_data_file_path`。对InnoDB主目录下表空间各组成文件的说明。这个选项的值是这些说明构成的一个列表，以分号间隔。每个文件的规格说明由文件名、文件长度和一些可能出现的选项组成，它们彼此以冒号间隔。InnoDB表空间各组成文件的总长度至少为10 MB。

如果这两个选项都没有设置值，InnoDB 存储引擎将在服务器的数据目录里创建一个初始长度是 10 MB、名字是 `idbata1` 的自扩展文件作为默认的表空间。通过使用这些选项，我们可以明确地对构成 InnoDB 共享表空间的文件的个数、长度和存放位置进行控制。

作为一个简单的例子，假设你打算在数据目录里创建了一个由两个长度都是 50 MB 的 `innodata1` 和 `innodata2` 文件构成的表空间。下面是这个表空间的配置情况：

```
[mysqld]
innodb_data_file_path = innodata1:50M;innodata2:50M
```

具体到这个例子，用不着对 `innodb_data_home_dir` 选项进行设置，因为它的默认值是 MySQL 服务器的数据目录，而你正想把 `innodata1` 和 `innodata2` 文件存放在那里。

InnoDB 存储引擎将按以下规则去组合 `innodb_data_home_dir` 和 `innodb_data_file_path` 选项的设置值以确定表空间文件的路径名。

- 如果innodb_data_home_dir选项的值为空，InnoDB存储引擎将把innodb_data_file_path选项里的所有文件规格说明解释为绝对路径名。空在这里的意思是你给出了那个选项但没有在等号后面给出一个值，而不是根本没有给出那个选项。
- 如果innodb_data_home_dir选项的值不为空，InnoDB存储引擎将把它解释为一个子目录的名字，然后再把innodb_data_file_path选项里的所有文件规格说明解释为这个子目录下的相对路径名。
- 如果innodb_data_home_dir选项根本就没有给出，它的默认值是MySQL数据目录的路径名，而InnoDB存储引擎将把innodb_data_file_path选项里的所有文件规格说明解释为该数据目录下的相对路径名。

根据上述规则，如果 MySQL 数据目录是/var/mysql/data，以下三组配置所指定的表空间文件就将是完全相同的：

```
[mysqld]
innodb_data_home_dir=
innodb_data_file_path=/var/mysql/data/ibdata1:50M;/var/mysql/data/ibdata2:50M

[mysqld]
innodb_data_home_dir=/var/mysql/data
innodb_data_file_path=ibdata1:50M;ibdata2:50M

[mysqld]
innodb_data_file_path=ibdata1:50M;ibdata2:50M
```

在设置 innodb_data_file_path 选项的时候，文件规格说明之间要用分号隔开，每个文件规格说明里的各组成部分要用冒号隔开。最简单的规格说明语法由一个文件名和一个文件长度值组成，但下列语法也是合法的：

```
path:size
path:size:autoextend
path:size:autoextend:max:maxsize
```

第一种格式给出的是一个长度固定为 size 的文件。size 必须是一个正整数，它后面还要有一个代表兆字节或千兆字节的字母 M 或 G。第二种格式给出的是一个可自动扩展文件，当这个文件快被填满时，InnoDB 存储引擎将自动地以递增方式扩展之。第三种格式与第二种类似，但增加了一个用来规定该文件最大长度的数字。只有表空间的最后一个组成文件是可自动扩展的。

在需要扩展表空间文件时，InnoDB 存储引擎将使用 8 MB 作为默认的递增量。如果想另行设定一个递增量，请设置 innodb_autoextend-increment 系统变量。

- InnoDB共享表空间的配置

在一般场合，共享表空间完全由常规文件构成，不包括任何未经格式化的硬盘分区（即设备文件）。以下是对一个完全由常规文件构成的共享表空间进行初始化设置的步骤。

- 把相应的语句添加到选项文件里。
- 确认用来存放表空间组成文件的子目录已经存在。InnoDB存储引擎只能创建有关的文件，不能创建子目录。
- 确认表空间组成文件都不存在。
- 启动MySQL服务器。InnoDB存储引擎将注意到有关文件尚不存在并把它们创建和初始化出来。

如果在未明确配置 InnoDB 存储引擎之前启动过 MySQL 服务器，就会存在一个由 InnoDB 存储引擎使用默认配置创建出来的共享表空间。对于这种情况，要想明确地重新配置表空间，必须先关停服务器并把与 InnoDB 有关的文件（表空间和日志文件）全部删除干净，再用你打算使用的配置选项重新启动服务器。（这一切必须在创建任何 InnoDB 数据表之前进行。如果已经创建了一些 InnoDB 数据表，必须在重新进行配置之前用 mysqldump 工具把那些数据表备份出来，等完成配置工作并重新启动服务器之后再恢复它们。）

使用未经格式化的硬盘分区作为 InnoDB 共享表空间的组成部分稍微麻烦一些，但有几个理由值得我们考虑这样做。

- 可以轻而易举地创建出一个非常巨大的表空间。由硬盘分区构成的表空间可以扩展到整个硬盘分区，而常规文件的最大长度却要受到操作系统的限制。
- 能保证整个存储空间的连续性，而常规文件却会受到文件系统碎片化的影响。为减少碎片化问题，InnoDB存储引擎在对表空间进行初始化的时候，会尽量向有关文件写入足够多的零去迫使操作系统把存储空间一次性地全部分配给它们而不是递增分配。这有助于减少碎片化问题，但不能从根本上杜绝。
- 可以减少文件系统管理层的开销。在某些系统上，这类开销可能不大，但在另一些系统上，这类开销可能会大到值得选用硬盘分区来构成表空间。

在考虑要不要使用硬盘分区来构成表空间时，还有一个因素很重要：有许多系统备份软件只针对文件系统，不能对硬盘分区进行备份，这意味着使用硬盘分区来构成表空间将会给系统备份工作增加困难。

用原始硬盘分区来构成表空间需要两个步骤。我们不妨假设你打算在一个 Unix 系统上使用路径名/dev/rdsk8 把一个 20GB 的原始硬盘分区分配给 InnoDB 表空间。此时，因为这个原始硬盘分区游离于 MySQL 数据目录之外，所以必须设定一个 innodb_data_home_dir 选项值。下面是具体的配置步骤。

(1) 在这个原始硬盘分区的长度值后面加上一个 newraw 后缀以表明这个文件其实是一个未格式化的硬盘分区，需要初始化：

```
[mysqld]
innodb_data_home_dir =
innodb_data_file_path = /dev/rdsk8:20Gnewraw
```

(2) 启动 MySQL 服务器。InnoDB 存储引擎将注意到 newraw 后缀并对这个分区进行初始化。它还将以只读方式去对待这个表空间，因为它知道你还没有完成第二个步骤。

(3) 完成了硬盘分区的初始化工作之后，关停 MySQL 服务器。

(4) 修改配置信息，把后缀 newraw 改为 raw：

```
[mysqld]
innodb_data_home_dir =
innodb_data_file_path = /dev/rdsk8:20Graw
```

(5) 再次启动 MySQL 服务器。因为那个后缀现在是 raw 而不是 newraw，InnoDB 存储引擎将明白这个硬盘分区已初始化，它就会以读/写方式去使用这个表空间了。

使用原始硬盘分区作为 InnoDB 表空间的一部分还要注意以下几个细节：首先，必须把这个硬盘分区的访问权限设置成允许 MySQL 服务器读/写；其次，必须保证这个硬盘分区没被用于其他目的，

否则如果有两个或更多个进程往这个分区上写数据，就会把事情弄得一团糟。比如说，如果错误地把系统数据交换分区用做了 InnoDB 表空间，系统迟早会发生崩溃！

在 Windows 系统上配置 InnoDB 共享表空间的时候，Windows 路径名中的反斜线字符既可以写成单个的斜线字符（“/”），也可以写成两个反斜线字符（“\\”）。此外，尽管在文件路径名里可能会出现冒号（完整的 Windows 路径名是以一个硬盘盘符和一个冒号开头的），你仍必须使用冒号来分隔各有关文件的规格说明。在遇到冒号的时候，InnoDB 存储引擎会查看它后面的字符并消除其二义性：如果后面的字符是一个数字，就表明那是某 InnoDB 组成文件的长度；否则，就说明那仍是一个路径名。请看下面这个配置，它创建的 InnoDB 表空间将由位于 C 盘和 D 盘上的两个长度分别为 50 MB 和 60 MB 的文件组成：

```
[mysqld]
innodb_data_home_dir =
innodb_data_file_path = C:/ibdata1:50M;D:/ibdata2:60M
```

在首次创建 InnoDB 表空间时，如果 MySQL 服务器因为 InnoDB 没能创建出某些个必要的文件而启动失败，请查阅 MySQL 服务器的“错误日志”以分析其原因。找出问题的根源之后，先删除 InnoDB 已经创建出来的所有文件（用来构成 InnoDB 表空间的原始硬盘分区不包括在内），再改正配置错误，最后重新启动 MySQL 服务器。如果使用了硬盘分区，千万不要忘记在对它进行初始化的时候需要使用 newraw 后缀，在启动并关停服务器之后要把它改回 raw。

- 重新配置InnoDB共享表空间

在对 InnoDB 共享表空间进行了初始化并开始使用它之后，就不能再改变它的组成文件的长度了。不过，MySQL 允许在现有的 InnoDB 表空间组成文件清单的末尾追加一个新文件，这一招在表空间快被数据填满时非常有用。InnoDB 表空间快被填满的征兆之一是一些本应该成功的 InnoDB 事务操作无法完成并且回滚。我们还可以用下面这条语句去了解还剩下多少可用空间，语句中的 tbl_name 可以是任何一个 InnoDB 数据表的名字：

```
mysql> SHOW TABLE STATUS LIKE 'tbl_name';
```

如果想通过增加一个新组成文件的办法来扩大 InnoDB 表空间，请按以下步骤进行：

(1) 关停正在运行的 MySQL 服务器。

(2) 如果 InnoDB 表空间的最后一个组成文件是可自动扩展的，就必须先把它改变为一个固定长度文件才能把另一个文件追加到它的后面。先查出这个文件的实际长度并把它换算成最接近的 MB 数（要按 1 MB = 1 048 576 个字节计算，不要按 1 MB = 1 000 000 个字节计算），再把计算结果写到它的规格说明里。比如说，假设 InnoDB 表空间由一个名为 ibdata1 的文件构成：

```
[mysqld]
innodb_data_file_path = ibdata1:100M:autoextend
```

如果这个文件现在的实际长度是 121 634 816 个字节，长度换算的结果就将是 121 634 816÷1 048 576 = 116 MB。所以要把它的规格说明改为下面这样：

```
[mysqld]
innodb_data_file_path = ibdata1:116M
```

(3) 把新的组成文件的说明追加到现有文件清单的末尾。如果新组成文件是一个常规文件，必须保证它此时尚不存在；如果新组成文件是一个原始硬盘分区，请按前面刚介绍过的两个步骤（先使用 newraw 后缀，在启动并关停服务器之后把它改回 raw）把它添加到 InnoDB 表空间里去。

(4) 重新启动 MySQL 服务器。

重新配置 InnoDB 共享表空间还有一种办法，即先备份它、再用新配置重新加载它。具体步骤如下。

(1) 使用 mysqldump 程序把保存在共享表空间里的 InnoDB 数据表全部备份出来。

(2) 关停 MySQL 服务器，删除现有的 InnoDB 共享表空间文件（硬盘分区不包括在内）、InnoDB 日志文件及对应于各 InnoDB 数据表的.frm 文件。（删除.frm 文件的另一个办法是在服务器运行时使用 DROP TABLE 语句来删除每一个 InnoDB 数据表。）

(3) 按新配置方案去重建 InnoDB 表空间。

(4) 把在第(1)步得到的备份文件重新加载到 MySQL 服务器里以重新创建各个 InnoDB 数据表。

- 让每个InnoDB数据表分别使用一个表空间

如果想让每个 InnoDB 数据表使用一个表空间，用--innodb-file-per-table 选项启动 MySQL 服务器即可。此后，每个 InnoDB 数据表将有一个.frm 格式文件和一个.ibd 数据文件，它们都存放在容纳着这个数据表的数据库的数据库目录里。

是否让每个 InnoDB 数据表分别使用一个表空间只影响 InnoDB 存储引擎在创建新数据表时的行为。不管 MySQL 服务器是不是用--innodb-file-per-table 选项启动的，InnoDB 存储引擎都能毫无问题地访问已经在共享表空间和各表空间里被创建出来的数据表。

2. 与InnoDB存储引擎有关的系统变量

前一节讨论了如何配置 InnoDB 的表空间。InnoDB 还有它自己的日志文件和内存缓冲区，还有几个其他的配置参数。下面是几个常用的 InnoDB 存储引擎配置参数：

- innodb_buffer_pool_size
  如果你有足够的内存，加大这个变量的值可以减少因为访问InnoDB数据表的数据和索引而引起的硬盘读写操作。
- innodb_log_buffer_size
  InnoDB存储引擎会尽量把关于每一个事务的信息缓冲在内存里，等整个事务结束时才一次性地把它们写到硬盘上。如果某个事务大到信息量超出了缓冲区的容量，就需要在它结束之前进行多次把缓冲区里的信息写到硬盘上去的操作。加大这个缓冲区的尺寸可以让更大的事务被缓冲在内存里而无需提前写入硬盘。这个变量的默认设置值是1MB，最大可取值是8MB。
- innodb_log_grou_home_dir
  InnoDB存储引擎有它自己的日志文件，它会在服务器每次启动时把尚不存在的日志文件创建出来。在默认的情况下，这些日志文件都将在数据目录里创建并以ib_作为文件名的前几个字符。innodb_log_grou_home_dir变量可以为InnoDB存储引擎指定一个用来存放其日志文件的子目录路径。但要特别注意的是，InnoDB存储引擎只能创建文件，不能创建子目录，所以在启动MySQL服务器之前一定要确认这个日志文件子目录已经存在。
- innodb_log_file_size和innodb_log_files_in_group
  当它的日志被写满时，InnoDB 存储引擎将把缓冲池里的信息写到磁盘上去。InnoDB 日志文件越大，就需要经过更长的时间才能被填满，需要把缓存池里的信息写到磁盘上去的次数也就越少。（不过，日志文件越大，数据库崩溃后的恢复工作就需要花费更多的时间才能完成。）改变 innodb_log_file_size 变量的值将改变 InnoDB 日志文件的长度，改变 innodb_log_files_in_group 变量的值将改变 InnoDB 日志文件的个数。InnoDB 日志文件的总长度（即 innodb_log_files_in_group 变量值和 innodb_log_file_size 变量值的乘积)是一个十分

重要的指标。InnoDB 日志文件的总长度不得超过 4 GB。

12.7.4　配置Falcon存储引擎

为了简化数据库的管理和维护工作，人们在设计 Falcon 存储引擎时为它增加了一些自我优化和免维护功能。比如说，Falcon 存储引擎的日志文件和表空间文件不仅可以在必要时自动扩展，还可以自动释放不再需要的可用空间。

在配置 MySQL 服务器时，可以通过下面几个参数来调控 Falcon 存储引擎的行为：

- `falcon_page_size`

  Falcon存储引擎使用固定的页面长度（默认值是4 KB）把数据写入表空间。我们可以通过为`falcon_page_size`赋值的办法来选择一个页面长度。它的可取值是1 KB、2 KB、4 KB、8 KB、16 KB或32 KB。必须在Falcon存储引擎尚未创建任何表空间文件之前对这个页面长度进行配置。如果想在Falcon已经创建表空间之后改变这个页面长度，必须备份并删除Falcon数据表、关停MySQL服务器并删除Falcon文件，然后使用新的页面长度重新启动MySQL服务器并重新加载备份文件。

- `falcon_serial_log_dir`

  Falcon存储引擎将在这个变量指定的子目录里创建它的日志文件。这个变量的默认值是MySQL服务器的数据目录。

Falcon 存储引擎允许创建“额外的”表空间来存放数据表里的数据。因此，我们可以把数据分散存储到多个物理设备上以获得更好的 I/O 性能。为 Falcon 数据表创建一个新的表空间需要使用 `CREATE TABLESPACE` 语句：

```
CREATE TABLESPACE myts
  ADD DATAFILE '/usr/local/mysql/data/falcon_myts.fts'
  ENGINE = FALCON;
```

在默认的情况下，Falcon 会在创建一个新数据表时为它分配一个默认表空间。如果想把一个数据表放到新的表空间里去，需要在相应的 `CREATE TABLE` 语句里使用一个 `TABLESPACE` 选项：

```
CREATE TABLE mytbl (i INT) ENGINE = Falcon TABLESPACE myts;
```

12.8　启用或者禁用 LOAD DATA 语句的 LOCAL 能力

`LOAD DATA` 语句的 `LOCAL` 能力不需要启用。它可以在编译或运行时用以下方法加以控制。

- 在MySQL服务器的编译阶段，可以在运行`configure`脚本时用`--enable-local-infile`或`--disable-local-infile`选项把客户库的`LOCAL`能力设置为默认启用或默认禁用状态。
- 在MySQL服务器的启动阶段，可以用`--local-infile`或`--skip-local-infile`选项来启用或者禁用MySQL服务器端的`LOCAL`支持。

12

如果在 MySQL 服务器端禁用了 `LOCAL` 能力，客户将根本不能使用这一能力。即使你在 MySQL 服务器端启用了 `LOCAL` 能力，MySQL 客户端库仍有可能把客户端的 `LOCAL` 能力设置为默认禁用状态，但某些客户程序可以在必要时激活它。比如说，`mysql` 程序有一个用来启用客户端 `LOCAL` 能力的 `--local-infile` 选项，`mysqlimport` 程序有一个`--local` 选项。

有些程序没有专门用来启用或禁用 `LOCAL` 能力的选项，但只要它们会读取选项文件，我们就仍可以控制这一能力。这取决于那些程序有没有用 `MYSQL_READ_DEFAULT_FILE` 或 `MYSQL_READ_`

DEFAULT_GROUP 选项调用过 mysql_option()函数。那两个选项的作用是让客户程序在连接 MySQL 服务器时读取选项文件。如果程序确实这样调用过 mysql_option()函数，我们就可以在一个适当的选项文件里列出一个--local-infile 或--disable-local-infile 选项来启用或者禁用 LOCAL 能力。对 mysql_option()函数的详细介绍见附录 G（需上网查阅）。

其他程序设计语言里的 MySQL 编程接口如果是基于 MySQL C API 的并且调用了 mysql_option()函数的话，它们也可以这样来控制 LOCAL 能力。比如说，在一个 Perl DBI 脚本里，我们可以在数据源的名字字符串里使用 mysql_ read_default_file 和 mysql_read_default_group 选项来控制脚本将如何连接 MySQL 服务器。

12.9　国际化和本地化问题

这里所说的“国际化”指的是软件能够在世界多个地区使用，“本地化”则指的是从软件的国际化支持里选择一个适用于本地区的来使用。与国际化和本地化有关的 MySQL 配置问题主要有以下几个方面：

- MySQL服务器默认使用的地理时区；
- MySQL服务器将使用何种语言来显示诊断信息和出错信息；
- 有哪些字符集可供MySQL服务器选用以及它选用的默认字符集。

12.9.1　设置MySQL服务器的地理时区

MySQL 服务器可以通过检查它所在的运行环境来设置一个默认的地理时区。一般来说，这将是服务器主机的本地时区。我们可以在启动 MySQL 服务器时为它明确地设定一个默认时区。在此基础上，MySQL 服务器还允许每一个连接成功的客户覆盖其默认设置而设置一个它自己的时区。这使应用程序可以根据客户程序的运行地点而不是 MySQL 服务器的运行地点来使用时间设置。下面的讨论描述了 MySQL 支持多个时区的能力。

MySQL 服务器把时区信息保存在两个系统变量里。

- system_ time_zone。服务器在启动时确定的服务器主机所在的地理时区。这个变量只有全局级变量一种形式，并且不允许在服务器运行期间重新设置。如果你想让MySQL服务器在启动时把system_ time_zone变量设置为你希望的值，在启动它之前把TZ环境变量设置为你希望的时区值即可。不过，在某些上下文里，让TZ环境变量能够记住你设置的值并不那么容易，比如当MySQL是作为整个系统开机启动过程的一部分而启动的时候。在Unix系统上，为MySQL服务器设置地理时区的另一个办法是使用--timezone选项配合mysqld_safe脚本（不能使用mysqld，因为它不支持这个选项）来MySQL服务器。建议大家把这个选项放到某个选项文件的[mysqld_safe]选项组里，尤其是在服务器通过不支持命令行选项的mysql.server脚本调用mysqld_safe脚本来启动的时候。比如说，如果想为mysqld_safe脚本设定美国中部时间，应该把下面这些代码添加到服务器的选项文件里：

  ```
  [mysqld_safe]
  timezone=US/Central
  ```

  上例中的语法适用于包括 Solaris、Linux 和 Mac OS X 在内的绝大多数 Unix 系统。另一种常见的语法是：

```
[mysqld_safe]
timezone=CST6CDT
```

请根据系统的具体情况去选用相应的语法。

- time_zone。代表着MySQL服务器的默认时区。在默认的情况下，这个变量被设置为SYSTEM，意思是使用system_ time_zone设置。在直接使用mysqld命令来启动MySQL服务器的时候，可以使用--default-time-zone选项来设置time_zone变量。服务器在运行时将使用全局级的time_zone值为每一个连接成功的客户设置它的会话级time_zone值，该值将成为客户的默认时区。任何一个客户都可以通过设置会话级time_zone变量的办法为它自己的连接重新设置一个时区。只有具备SUPER权限的管理性客户（程序）才可以设置全局级time_zone变量，这将影响此后连接到服务器的所有客户的默认时区。

下面这条语句可以查出全局级和会话级时区变量的当前值：

```
SELECT @@GLOABL.time_zone, @@SESSOIN.time_zone;
```

可以用 3 种值来设置 time_zone 变量，但其中一种需要额外的管理性操作。下面给出的语句对会话级 time_zone 变量进行了设置。如果是具备 SUPER 权限的用户，只要把下面这些语句中的 SESSION 替换为 GLOBAL 就可以对全局级 time_zone 变量进行设置了。

- 把SYSTEM赋值给time_zone变量，这将把它设置为system_ time_zone变量的值：

```
SET SESSION time_zone = 'SYSTEM';
```

- 把一个带正负号的“小时加分钟”时间值赋值给time_zone变量，这是相对于UTC标准时间的偏移值：

```
SET SESSION time_zone = '+00:00';     # UTC
SET SESSION time_zone = '+03:00';     # 3 hours ahead of UTC
SET SESSION time_zone = '-11:00';     # 11 hours behind UTC
```

- 把一个地理时区名赋值给time_zone变量，这将把它设置为给定地理时区：

```
SET SESSION time_zone = 'US/Central';
SET SESSION time_zone = 'CST6CDT';
SET SESSION time_zone = 'Asia/Jakarta';
```

要想使用最后一种办法（使用地理时区的名字来进行设置），必须先让 MySQL 服务器理解那些地理时区的名字，这需要从操作系统的时区文件里把有关信息加载到 mysql 数据库里的一组数据表里。MySQL 软件的安装过程不会自动完成这项工作。我们需要以手动方式运行 mysql_tzinfo_to_sql 程序去填充那些数据表，这个程序将读取操作系统的各个时区文件并根据它们的内容生成相应的 SQL 语句。把这些语句馈入 mysql 程序以执行它们。

如果你的系统上有时区文件，建立时区数据表的第一步是确定它们安装在什么地方。假设这个地方是/usr/share/zoneinfo 子目录，把时区文件加载到 mysql 数据库的语句将是下面这样：


```
% mysql_tzinfo_to_sql /usr/share/zoneinfo | mysql -p -u root mysql
```

然后重新启动服务器。这个办法适用于绝大多数 Unix 版本。如果你的系统是 Windows 系统或是没有时区文件的 Unix 系统，你将需要从下面这个网址下载一组预先建立好的 MyISAM 数据表，它们的内容是时区信息：

```
http://dev.mysql.com/downloads/timezones.html
```

对下载回来的文件进行解压缩。在 MySQL 服务器没有运行的情况下，把解压缩得到的.frm、.MYD 和.MYI 文件复制到 MySQL 数据目录下的 mysql 数据库子目录里。然后重新启动服务器。

12.9.2　选择用来显示出错信息的语言

MySQL 服务器能够用好几种语言来显示诊断信息与出错信息。它的默认值是 english（英语），但可以设置其他语言。如果想知道都有哪几种语言可供选用，请查看 MySQL 安装路径下的 share/mysql 子目录里有几个以语言名称作为名字的下级子目录。如果想改用另一种语言，请用--language 启动选项给出该种语言的名称或路径名。以法语为例，如果 MySQL 的安装目录是/usr/local/mysql，就要使用--language=french 或--language= /usr/local/mysql/share/mysql/french 选项来改用法语显示有关信息。

12.9.3　配置MySQL服务器的字符集支持

字符集决定着哪些字符允许用在字符串值里。MySQL 支持多种字符集，我们可以在服务器、数据库、数据表、数据列和字符串常数等级别选用字符集。MySQL 还为每种字符集支持多种排序方式。排序方式将影响到字符串的比较和排序操作。

本小节的讨论重点是如何配置 MySQL 服务器的字符集支持。第 2 章概括了 MySQL 服务器的字符集支持能力。对字符串数据列的创建和使用方法的讨论见第 3 章。

从源代码开始建立 MySQL 服务器时，可以对 MySQL 服务器将支持哪几种字符集以及它将默认使用的字符集和排序方式做出设定，这需要用到 configure 脚本和以下选项：

- 如果想增加MySQL服务器支持的字符集，就要使用--with-extra-charsets选项。这个选项的参数是一个以逗号为分隔符的字符集名单。比如说，如果想让MySQL服务器支持latin1、big5和hebrew字符集，就要像下面这样去配置MySQL软件的源代码发行版本：

  ```
  % ./configure --with-extra-charsets=latin1,big5,hebrew
  ```

  --with-extra-charset 选项有两个特殊的参数值：all 代表所有的可用字符集，complex 代表所有的复杂字符集。所谓“复杂字符集”包括多字节字符集和有特殊排序规则的字符集。如果想知道都有哪些字符集可以选择，请使用如下所示的命令并在它的输出里寻找对--with-charset 选项的描述：

  ```
  % ./configure --help
  ```

- MySQL服务器的默认字符集是latin1，但可以用--with_charset选项另行指定一个。
- MySQL服务器的默认排序方式是latin1_swedish_ci，但可以用--with_collation选项另行指定一个。排序方式必须与默认字符集兼容。（直观地讲，就是排序方式的名字必须以相应的字符集的名字开头。）

下面这条配置命令用到了上面介绍的 3 个选项：

```
% ./configure --with-charset=utf8 \
    --with-collation=utf8_icelandic_ci \
    --with-extra-charsets=all
```

在默认的情况下，MySQL 服务器在启动时将把它的默认字符集和排序方式设置为在编译阶段设定的默认值，但可以用--character-set-server 和--collation-server 启动选项另行指定一个默

认值。排序方式必须与字符集兼容。

在运行一个客户程序的时候，可以用`--default-character-set` 选项告诉客户程序使用哪个字符集。如果指定的字符集没被安装在它的默认位置，但可以在另一个子目录里找到所需要的字符集文件，还可以用`--character-set-dir` 选项把那些文件的存放位置告知客户程序。

12.10 运行多个服务器

多数人在一台机器上能够运行多个 MySQL 服务器，但是要有能够用来运行多个服务器的环境。

- 你想要测试服务器的新版本，同时要使你的服务器仍然运行。在这种情况下，应该运行不同的服务器二进制文件。
- 你想要试试复制（replication）机制，但你只有一个服务器主机，并且必须在同一台机器上运行主服务器和从服务器。
- 操作系统通常在打开文件描述符的数量上对每个进程加以限制。如果你的系统难于增加极限值，运行服务器二进制文件的多个实例是解决这一问题的一种方法。（例如，增加极限值或许需要重新编译内核，而且你如果不负责管理这台机器时还不能做这项工作。）
- 因特网服务提供者通常给各个用户提供他们自己的MySQL设置，这就必须要多个服务器。这在所有用户运行同一版本的MySQL时可能要运行同一二进制文件的多个实例，或者如果有用户运行和其他人不同的版本，则也要运行不同的二进制文件。

以上是一些最普通的运行多个服务器的理由，当然还有其他原因。例如，如果你需要写入 MySQL 文档，通常需要测试各种服务器版本，查看它们的工作情况有何不一样。归入这一类的理由是安装了许多服务器。然而，总共只运行了少量几个，其他的仅在测试时运行，所以必须能按要求方便地启动和停止它们。

12.10.1 运行多个服务器的问题

运行多个服务器比运行一个要复杂，因为必须防止它们互相干扰。某些问题可能是安装 MySQL 时出现的。如果你使用不同的版本，它们必定会各自放入不同的位置。对于预编译的二进制分配，你可能把它们分割成不同目录来实现。对于你自己编译的源发行版本，可以使用配置的`--prefix` 选项为每个分配指定不同的安装位置。

其他问题是在启动服务器运行时发生的。每个服务器进程必须有几个专用的参数值。例如，每个服务器必须监听进程连接用的不同的 TCP/IP 端口，否则它们会相互冲突，无论你运行不同的服务器二进制文件还是相同二进制的多个实例。对于其他连接接口也是如此，如 Unix 套接字文件、Windows 命名管道或共享内存。如果你启用日志，每个服务器必须写至自己一组的日志文件，因为让不同的服务器写至相同的文件中一定会产生问题。

可以在启动服务器运行时指定服务器的选项，一般在选项文件中。另一种方法是，如果运行自己从源编译的几个服务器二进制文件，则可以在建立进程期间为每个服务器指定一组不同的参数值。这些成为其内部的默认值，而且不必在运行时明确指出它们。

运行多个服务器时，一定要牢牢记住使用的参数，以便你不会忘记发生的事情。进行这项工作的一种方法是使用指定该参数的选项文件。即使服务器有编译进的专用参数值，这也是有用的，因为选项文件用作一种文档。

下列讨论枚举了几种选项，如果它们不是依据每个服务器进行设置，就有可能引起冲突。注意，某些选项会影响其他选项，这样，不必为每个服务器明确设置每一个选项。例如，每个服务器必须在运行时使用一个专用的日志文件。但是数据目录是所有日志文件用的默认地点，所以当每个服务器具有一个不同的数据目录时，会隐式导致不同的日志文件。

- 如果你运行不同的服务器版本，通常每个发行版本安装在不同的基目录下。每个服务器应该使用不同的数据目录。（Windows上强制使用不同的数据目录，Unix上强烈推荐使用不同的数据目录。）为了明确地指定这些值，使用下列选项：

选项	用途
`--basedir=`*dir_name*	至MySQL安装根目录的路径名
`--datadir=`*dir_name*	至数据目录的路径名

  在许多情况下，数据目录是基目录的一个子目录，但不一定总是这样。例如，因特网服务提供者可以为其用户提供一个公用的 MySQL 服务器和客户程序，但是运用不同的服务器实例，每一个实例管理给定用户的数据目录。在这种情况下，基目录对于所有服务器是一样的，但各自的数据目录可以放在不同地方，或许在用户的主目录下。

- 下列选项必须有不同的数值供每个服务器用，要防止服务器相互改变：

选项	用途
`--port=`*port_num*	TCP/IP连接用的端口号
`--socket=`*file_name*	至Unix区域套接字文件的路径名或Windows命名管道名称
`--pid-file=`*file_name*	至服务器写入其进程ID的文件的路径名
`--shared-memory-base-name=`*name*	用于共享内存连接的共享内存的名字（只限于Windows）

  在 Windows 上，`--socket` 或`-shared-memory-base-name` 选项需要仅提供给有`--enable-named-pipe` 或`--shared-memory` 选项的服务器，以启用命名管道或共享内存连接。这时，有一个服务器可以使用默认命名管道和共享内存名称（分别为 MySQL 和 MYSQL），但其他服务器则必须指定不同的名称。

- 如果你启用日志，使用的任何日志名对每一个服务器一定要是不相同的。不然的话，会有多个服务器争夺记录至同一个文件中。那是最混乱的局面，最坏的情况会阻碍复制等操作的正确工作。由下表中的选项命名的日志文件建立在服务器数据目录下，只要你指定相对文件名即可。如果每个服务器使用不同的数据目录，不必指定绝对路径名使每一个记录至不同组的文件中（参见本章12.5节中有关命名日志文件的内容）。

日志文件选项	选项许可的文件
`--log-error[=`*file_name*`]`	错误日志文件
`--log[=`*file_name*`]`	一般日志文件
`--log-slow-queries[=`*file_name*`]`	慢查询日志文件
`--log-output[=`*destination*`]`	一般/慢查询日志目标
`--log-bin[=`*file_name*`]`	二进制日志文件
`--log-bin-index=`*file_name*`]`	二进制日志索引文件
`--relay-log[=`*file_name*`]`	回复日志文件
`--relay-log-index=`*file_name*	回复日志索引文件

如果启用 BDB 或 InnoDB 数据表处理机，它们写入日志的目录必须是每个服务器专用的。服务器默认地把这些日志写入数据目录中。可以使用--masfer-info-file 和--relay-log-info-file 选项进行设置。

- 在Unix下，如果你使用mysql_safe启动服务器，则会建立一个错误日志文件（默认在数据目录中）。你可以明确地用--err_log=*file_name*指定错误日志名。但是，在MySQL 5.1.11之前，如果指定一个相对路径，mysqld_safe将解释为是相对于mysqld_safe调用的目录，并不是相对其他数据目录。如果你指定一个绝对路径名，就能保证你一定在同一地点建立错误日志。从MySQL 5.1.20起，可以将错误输出发送到syslog。详情参见12.5.1节。
- 如果启用了InnoDB或Falcon存储引擎，它写日志的目录对于每个服务器而言必定是唯一的。默认情况下，这些引擎将其日志写入数据目录。为了改变位置，使用下表中的选项。

日志选项	目的
--innodb_log_group_home_dir=dir_name	InnoDB日志文件目录
--falcon-serial-log-dir=dir_name	Falcon日志文件目录

每个使用 InnoDB 的服务器必须被配置以使用它自己的共享表空间，详细内容可参见 12.7.3 节的第 1 小节。

- 在Unix下，也有必要以每个服务器为基础指定--user选项，指出运行每个服务器要用的注册账号。就好像你为不同用户提供了各个MySQL服务器实例，每个用户“拥有”一个独立的数据目录。
- 在Windows下，以服务安装的不同服务器每一个都必须使用一个不同的服务名。

12.10.2 配置和编译不同的服务器

如果我们打算建立不同版本的服务器，应该把它们安装在不同的地方。保持不同版本相互隔开的最容易的方法是在运行 configure 时使用--prefix 选项，为每一个版本指示一个不同的安装基目录。如果你把版本号加入基目录名中，这就便于告知哪个目录对应于 MySQL 的哪个版本。本节讨论了一种实现上述目的的方法，叙述了特定的安装约定，用来保持自己的 MySQL 安装设置隔离。

我把所有 MySQL 设置放在一个公共目录中：/var/mysql。为了安装给定的发行版本，使用发行的版本号把该发行版本放在名叫/var/mysql/的子目录内。例如，我使用/var/mysql/50124 作为 MySQL 5.1.24 的安装基目录，这可以靠运行 configure 用--prefix=/var/mysql/50124 来实现。也可使用其他选项，用特定于服务器的值，例如 TCP/IP 端口号和套接字路径名来实现。使用的配置可使 TCP/IP 端口号等于版本号，把套接字文件直接放入基目录中，并以 data 命名数据目录。

为了设置这些配置选项，使用一个名叫 config-vre 的 shell 脚本程序，如下所示（注意，configure 用的选项是--localstatedir，不是--datadir）：

```
VERSION=50124
BASEDIR=/var/mysql/$VERSION
TCP_PORT=$VERSION
HANDLERS="
  --with-archive-storage-engine
  --with-csv-storage-engine
  --with-federated-storage-engine
  --with-innodb
```

```
"
OTHER="
  --enable-local-infile
  --with-embedded-server
  --with-extra-charsets=all
  --with-partition
  --with-row-based-replication
  --with-ssl
"
rm -f config.cache
CXX=gcc \
./configure \
  --prefix=$BASEDIR \
  --localstatedir=$BASEDIR/data \
  --with-unix-socket-path=$BASEDIR/mysql.sock \
  --with-tcp-port=$TCP_PORT \
  $HANDLERS $OTHER
```

要保证第一行设置正确的版本号，按照要编译进来的存储引擎，以及是否为 LOAD DATA 启用 LOCAL 支持等因素来修改其他数值。完成后用下列命令进行配置，建立和安装该发行版本：

```
% sh config-ver
% make
% make install
```

这些命令用于已打包的源发行版本，如果要使用最新的开发源，必须像 MySQL 用户手册中描述的那样在可以使用 config_ver 之前创建 configure 脚本。

安装好 MySQL 给定版本后，把当前位置改成安装的基目录，并初始化数据目录和权限表：

```
# cd /var/mysql/50124
# ./bin/mysql_install_db --user=user_name
```

user_name 是登录账户的名字（如 mysql 账户），用于运行服务器。在以 root 或 user-name 登录时应运行这些命名。

在这一点上，执行 MySQL 安装目录锁定步骤，这已在 12.2.1 节的第 1 小节中简要叙述，更详细的内容在 13.1.2 节中叙述。

其后，剩下的工作就是设置选项文件中要用的任何选项和安排启动服务器。12.10.4 节将讨论进行这项工作的一种方法。

12.10.3　指定启动选项的决策

安装服务器以后，如何使用每个服务器运行时需要的正确选项来启动你的服务器？有下列几种选择。

- 如果你运行由你自己建立的不同服务器，你可以对每一个服务器在一组不同的默认值里进行编译，而且在运行时不必给定选项，其缺点是给定服务器使用什么样的参数不一定是唯一的。
- 为了指定在运行时的选项，可以在命令行或选项文件内列出它们。如果你必须指定许多选项，把它们写在命令行上似乎是不实际的。把它们放在选项文件内更方便，尽管该技巧是为了每个服务器能读取正确的选项设定。实现这个目的的决策包括下列工作。
  - 使用--defaults_file选项指定服务器要读的文件，以便找出它的全部选项，并为每个服务器指定一个不同的文件。这样，可以把给定服务器需要的全部选项放入一个文件中，以便配置全部在一个地方指定。（注意，当你使用该选项时，服务器将不会读平常的选项文件，例

如/etc/my.cnf)。

- 把所有服务器共用的选项放入全局性选项文件中，例如/etc/my.cnf。使用命令行上的--defaults_extra_file选项指定一个含有专用于给定服务器的附加选项文件。例如，为应用于全部服务器的选项使用/etc/my.cnf中的[mysqld]组。这些要求不必在每个服务器的选项文件中重复。

  一定要使运行的所有服务器懂得放在共用选项组内的任一选项。例如，当任一服务器比版本 5.1.6 还老时，不能使用 event_manager=1 来启用事件调度程序，因为这已引入了那个选项，其在共用选项文件中的出现会使较老的服务器产生启动故障。

  如果所有服务器都为 MySQL 4.0.2 或更新的版本，你可以使用 loose_opt_name 语法来指定所有服务器都不懂的选项。如果服务器不理解以这种方式给出的选项，它将忽略这个选项，在得到警告后继续执行。有关 loose 选项的信息，参见 F.1 节。

- 使用mysqld_multi脚本程序来管理多个服务器的启动。这个脚本程序允许你在一个文件中列出所有服务器用的选项，但是每个服务器应和文件中自己特定的选项组相结合。

- 在Windows下，你可以运行多个服务器，使用专门的选项文件组命名约定，专用于这种服务器配置。参见12.2.2节的第2小节。

下面几节给出了采用这些决策的某些方法，说明如何使用 mysqld_multi 和如何在 Windows 下运行多个服务器。

12.10.4　用于服务器管理的**mysqld_multi**

在 Unix 上，mysqld_safe 和 mysqld_server 脚本程序通常用于启动服务器，在单个服务器设置中这两个工作是最好的。为了便于处理几个服务器，可以使用 mysqld_multi 脚本程序来替代。

mysqld_multi 根据你给每个你要建立的服务器配置赋予的专用号码进行工作，然后把服务器选项列在一个选项文件组[mysqldn]内，其中 n 就是那个号码。选项文件也可包括一组[mysqld_multi]，它列出专用于 mysqld_multi 自己的选项。例如，如果服务器装有 MySQL 5.0.56，5.1.24 和 6.0.5，则[mysqld50056]，[mysqld50124]和[mysql60005]指定它们的选项组，并且把这些选项设置在/etc/my.cnf 文件中，如下所示：

```
[mysqld50056]
basedir=/var/mysql/50056
datadir=/var/mysql/50056/data
mysqld=/var/mysql/50056/bin/safe_mysqld
socket=/var/mysql/50056/mysql.sock
port=50056
user=mysql
log=qlog
log-bin=binlog
innodb_data_file_path = ibdata1:100M

[mysqld50124]
basedir=/var/mysql/50124
datadir=/var/mysql/50124/data
mysqld=/var/mysql/50124/bin/mysqld_safe
socket=/var/mysql/50124/mysql.sock
port=50124
user=mysql
```

12

```
log=qlog
log-bin=binlog
innodb_data_file_path = ibdata1:50M:autoextend
event_scheduler=ON

[mysqld60005]
basedir=/var/mysql/60005
datadir=/var/mysql/60005/data
mysqld=/var/mysql/60005/bin/mysqld_safe
socket=/var/mysql/60005/mysql.sock
port=60005
user=mysql
log=qlog
log-bin=binlog
skip-innodb
language=french
character-set-server=utf8
```

为每个服务器设定的配置参数对应于该目录配置，这在 12.10.2 节中已讨论过。同时还指定了特定于其他服务器的参数，其对应于不同类型的日志、存储引擎等。

为了启动给定的服务器，在命令行上调用具有命令字 start 和服务器选项组编号的 mysqld_multi：

```
% mysqld_multi --no-log start 50124
```

--no-log 选项使得状态信息发至终端，而不是发至日志文件。你可以查看什么更容易执行。可以指定多个服务器，只要用以逗号分隔的列表给定组号即可。服务器编号的范围是由短划线分隔开号码来指定的。然而，在服务器列表中一定没有空白区：

```
% mysqld_multi --no-log start 50056,50124-60005
```

为了停止服务器或得到服务器是否运行的状态报告，使用一个命令字 stop 或 report，其后紧跟服务器列表。对于这些命令，mysqld_multi 将调用 mysqladmin 和该服务器通信，所以还必须指定一个用户名和管理账号用的口令：

```
% mysqld_multi --nolog --user=root --password=rootpass stop 50056
% mysqld_multi --nolog --user=root --password=rootpass report 50056,60005
```

用户名和口令必须适用于你要用给定命令控制的所有服务器。mysqld_multi 试图自动确定 mysqladmin 的位置，你也可以明确指定选项文件[mysqld_multi]组的路径。还可以列出用于 stop 和 report 命令的选项组内的默认管理用户名和口令。例如：

```
[mysqld_multi]
mysqladmin=/usr/local/mysql/bin/mysqladmin
user=leeloo
password=multipass
```

出于安全考虑，最好把管理口令放在一个选项文件中，而不是在命令行暴露它们。如果你把口令放在一个文件中，要保证其不能公开可读！参见 13.1.2 节的第 2 小节，了解如何让文件私有化。

12.10.5　在Windows系统上运行多个MySQL服务器

在 Windows 系统上运行多个 MySQL 服务器有两个方法：一是以手动方式依次启动它们，二是运

行多个 Windows 服务。如果你愿意，还可以把这两种办法结合起来使用。

如果决定以手动方式启动多个 MySQL 服务器，需要为它们分别创建一个选项文件并把它们各自的参数写到相应的文件里去。比如说，如果想运行同一个 MySQL 服务器程序的两个实例但让它们各有各的数据目录，需要创建如下所示的两个选项文件：

C:\my.ini1 文件：

```
[mysqld]
basedir=C:/mysql
datadir=C:/mysql/data
port=3306
```

C:\my.ini2 文件：

```
[mysqld]
basedir=C:/mysql
datadir=C:/mysql/data2
port=3307
```

在启动某给定 MySQL 服务器之前，必须先把它的数据目录创建好，这是因为供 Windows 系统使用的 MySQL 发行版本没有提供与 mysql_install_db 脚本功能相似的工具。具体到这里的例子，要想把 MySQL 安装到 C:\mysql 子目录里，必须提前把 C:\mysql\data 子目录创建好，而建立 C:\mysql\data2 子目录的最简单的办法是使用如下所示的命令把它创建为 C:\mysql\data 子目录的一个副本（在没有运行任何 MySQL 服务器的情况下）：

```
C:\> xcopy C:\mysql\data C:\mysql\data2 /E
```

然后，在从命令行启动 MySQL 服务器的时候，用--defaults-file 选项来表明你想使用哪一个选项文件：

```
C:\> mysqld --defaults-file=C:\my.ini1
C:\> mysqld --defaults-file=C:\my.ini2
```

这两个 MySQL 服务器启动之后，客户程序——包括用来关停 MySQL 服务器的 mysqladmin 程序在内——需要通过指定端口才能建立与它们的连接。下面的第一条命令使用的是默认端口（3306），而第二条命令明确地指定了端口 3307：

```
C:\> mysqladmin -p -u root shutdown
C:\> mysqladmin -p -u root -P 3307 shutdown
```

如果想把 MySQL 服务器安装为一项 Windows 服务，需要使用--install 选项。比如说，下面两条命令都可以把 mysqld 安装为一项服务：

```
C:\> C:\mysql\bin\mysqld --install
C:\> C:\mysql\bin\mysqld --install service_name
```

--install 命令必须使用 MySQL 服务器的完整路径名。如果没有给出 service_name 参数或给出的是 MySQL，则表示使用默认的服务名（即 MySQL）；否则，使用给定的名字作为服务名。（上面两条命令将使 MySQL 服务器去读取不同的选项组，有关规则参见 12.2.2 节的第 2 小节。）

我们来看一个例子。假设你想运行两个 mysql 实例并让它们分别使用 MySQL 和 mysqlsvc2 作为服务名，分别使用名为 MYSQL 和 mysqlsvc2 的共享内存，而 MySQL 数据目录的布局仍和前面例子里一样。在某个标准的选项文件（比如 C:\my.ini）里为这两个 MySQL 服务器设置如下所示的选项：

12

```
# group for default ("MySQL") service
[mysqld]
basedir=C:/mysql
datadir=C:/mysql/data
port=3306
shared-memory
shared-memory-base-name=MYSQL

# group for "mysqlsvc2" service
[mysqlsvc2]
basedir=C:/mysql
datadir=C:/mysql/data2
port=3307
shared-memory
shared-memory-base-name=mysqlsvc2
```

选项组的先后顺序非常重要。安装在默认服务名 MySQL 下的服务器将只读取[mysqld]选项组，但安装在非默认服务名 vmysqlsvc2 下的服务器将读取[mysqld]和[mysqlsvc2]两个选项组。通过在选项文件里把[mysqlsvc2]选项组放在[mysqld]选项组后面，就可以确保[mysqlsvc2]选项组里的选项将覆盖[mysqld]]选项组里的对应选项所做出的设置，从而正确地把第二个 mysql 实例运行为 mysqlsvc2 服务。安装和启动这些服务的命令如下所示：

```
C:\> C:\mysql\bin\mysqld --install
C:\> net start MySQL
C:\> C:\mysql\bin\mysqld --install mysqlsvc2
C:\> net start mysqlsvc2
```

安装服务器时，如果给出了一个服务名，还可以用一个--defaults-file 选项作为这个命令行上的最后一个选项：

```
C:\> C:\mysql\bin\mysqld --install service_name --defaults-file=file_name
```

这提供了另一种设定 MySQL 服务器专用选项的手段。MySQL 服务器将记住用--defaults-file 选项给出的文件名并在启动过程中读取其内容，MySQL 服务器将从这个文件的[mysqld]选项组去读取选项。

在有多个 MySQL 服务器同时运行的时候，客户可以使用默认的 TCP/IP 端口或是共享内存名去连接默认的服务器。如果想连接到第二个服务器，必须明确地给出它的 TCP/IP 参数或共享内存参数，如下所示：

```
C:\> mysql --protocol=tcp --port=3307
C:\> mysql --protocol=memory --shared-memory-base-name=mysqlsvc2
```

可以用 mysqladmin shutdown 命令、net stop 命令或 Services Manager（服务管理器）来关停服务器。如果想卸载服务器，先关停（如果它仍在运行的话），然后用--remove 命令字和你在安装服务器时使用的服务名来卸载它。如果是默认的服务名 MySQL，则可以省略服务名。如下所示：

```
C:\> mysql --remove
C:\> mysql --remove mysqlsvc2
```

12.11 升级 MySQL

MySQL 一直处于开发阶段，常常会升级。于是对 MySQL 管理员提出了问题，当新的版本出现时你是否要升级现有的 MySQL。本节提供一些指导帮助你作出决断。

当新的版本出现时你应该做的第一件事是要找出它和前一版本有什么不同。为了保证你知道新的版本，订阅 `announce@lists.mysql.com` 发送的邮件列表（访问 http://lists.mysql.com/）。每个通告包括新的修改注意项，所以这是个很好的办法，保证获悉新的开发版本。（另一种办法是查看《MySQL 参考手册》中的 Release Note 或 Change Notes，自己来了解什么算新的。）还应阅读参考手册有关升级的内容，了解相关发行系列。那一节指出了应注意的所有重要问题，以及升级时必须遵守的步骤。如果新版本引入了与旧版本不兼容的内容，这些信息尤为重要。

- 你使用新版本修正过的当前版本碰到什么问题没有?
- 新版本是否有你想要或需要的附加性能?
- 新版本是否改进了你使用的某些类型的操作性能?

如果所有这些问题的回答都是 No，则没有任何令人信服的理由进行升级。如果其中任一个问题的答案为 Yes，则可以继续进行。在这一点上，有用的方法往往是等几天再查看 MySQL 发送的邮件列表，看看他人对该版本发表了什么意见。升级有帮助吗？存在 bug 或其他问题吗？

考虑其他一些因素有助你作出决断，如下所述。

- 稳定系列中的版本最常用于修正bug，而不是添加新性能。在稳定系列内修改比在开发系列中修改的风险一般小一点。
- 有可能当你升级MySQL时，必须升级已链接的MySQL C客户库建立的其他程序。例如，MySQL 升级后，还必须新构建其他库或依赖MySQL C客户库的应用程序，以便把新的客户库链接上。（包括Perl DBD::mysql模块和PHP。当你升级MySQL之后，你所有和MySQL有关的DBI和PHP脚本程序启动主存储器信息的转储，这是你必须重建它们的明显征兆。）如果你不愿重建，那么最好不要升级MySQL。如果使用静态链接而不是动态链接的程序，发生该问题的可能性会大大降低。然而，这样会增加系统存储器的需求量。

如果还不确定是否升级，独立于现有的服务器对新的服务器进行测试。测试既可以和工作的服务器并行，也可以安装在不同的机器上。如果使用不同的机器，则易于保持服务器之间的独立性，因为你有更大的自由度进行配置。如果你选择了新的服务器和现有的服务器在同一主机上并行，一定要用专用的参数值来配置，例如安装位置、数据目录、服务器监听连接的网络接口。参见 12.10 节的相关内容。

在另外一种情况下，你或许要用现有数据库中的数据副本来测试新的服务器。参见 14.3 节中对复制数据库的说明。

如果升级到不能和老版本兼容的版本，然后决定转回到较早版本的话，这可不是如此容易降级的。例如，MySQL 5.0 的早期版本有一些变化（`VARCHAR` 和 `DECIMAL` 数据类型的存储格式就是典型例子）。如果从 MySQL 4.1 升级至 5.0 或更高版本，而且把你的数据表转换成 5.0 格式，这将和 4.1 服务器不兼容。如果你决定降级至 4.1 版，有一个好主意是使用 `mysqldump` 的`--compatible` 选项转储数据库，生成转储文件，加载到老版本。

不要害怕试用开发版本

为了生产目的，例如管理你的资产，明智的做法是不用开发版本。另一方面，我的确想鼓励你用你生产数据的一个副本来测试新版本。试用新版本的人越多，他们的实践经验越丰富。这样就更容易找出可能存在的 bug，这是件很好的事情。bug 报告是帮助 MySQL 开发向前发展的重要因素，因为开发者实际上在修正用户社区报告的问题。

如果想有一个数据来源供试验服务器执行查询，考虑用一个生产服务器作为复制的主服务器，而设置试验服务器为复制的从服务器。这样，主服务器执行的修改将发往从服务器，给它提供连续的输入。主服务器不会将任何检索信息发送给从服务器，但你可以在从服务器上指定客户程序，发出 SELECT 语句来查看如何处理它们。

第 13 章

访问控件和安全

作为一个系统管理员，你有责任保证数据库内容的安全，只让拥有权限的人访问数据，为此，你必须维护 MySQL 设置的安全性和完善性。第 12 章已经触及一些有关安全性的话题，例如设置初始 MySQL `root` 用户口令的重要性和如何设置用户账号。这些是你设置启动和运行过程的一部分。在本章中，你将会看到和安全性更加密切的几个方面。

- 为什么安全性很重要，你应当防护哪些破坏。
- 面对服务器主机上具有注册账号的其他用户可能带给你的风险（内部安全性）你能做些什么。
- 面对网络上连接至服务器的客户可能带给你的风险（外部安全性）你能做些什么。

内部安全性指的是与能够直接访问 MySQL 服务器主机的其他用户——那些在 MySQL 服务器主机上有登录账户的其他用户——有关的问题。一般来讲，内部安全漏洞大都与文件系统访问有关，你需要保护你的 MySQL 数据库系统免遭在 MySQL 服务器主机上有账户的人们的攻击。尤其是服务器的数据目录，只有你用来运行 MySQL 服务器的那个登录账户才能拥有和控制。如果做不到这一点，在安防方面的其他努力就可能白费。比如说，经网络连接到 MySQL 服务器的客户的访问权限是由权限表来控制的，你应对权限表里列出的账户进行正确的设置，但那些表的完整性取决于合适的文件系统保护。如果你把你数据目录内容的访问权限设置得过于宽松的话，只要替换掉相应的权限表文件，别人就能轻易地改变你所设置的客户访问控制策略。

外部安全性涉及从外部连接的客户问题。必须保护 MySQL 服务器免遭通过网络进入的连接要求访问数据库内容时可能带来的危险。你应该设置 MySQL 权限表，不允许访问由服务器管理的数据库，除非提供有效的姓名和口令。另一种危险是可能有第三者监视着网络并俘获服务器和客户之间的通信联系。就此而论，你或许要配置你的 MySQL 装置。支持使用安全套接字层（SSL）协议的连接。

本章介绍了这些主题，你应该了解和发出指令，如何在内、外两级防止未经授权的访问。本章会经常提及用于运行 MySQL 服务器和执行其他和 MySQL 有关的管理任务的注册账号。文中对该账号所用的用户名和组名都是 `mysql`。如果你使用其他的用户名和组名(例如,使用自己的账户运行 MySQL 服务器)，则改变其名称。

13.1　内部安全性：防止未经授权的文件系统访问

本节叙述如何锁住 MySQL 装置，使其免遭服务器主机上未经授权的用户的破坏。本节仅适用于 Unix 系统。这里假设你是在 Windows 上运行服务器，对机器有完全的控制权，并且没有其他的本地用户。

MySQL 安装过程建立了几个目录，有一些需要不同的保护。例如，不需要服务器程序能被 MySQL 管理账号以外的任何人访问。相反，客户程序通常是公开可以访问的，所以其他用户可运行它——但不能修改或更换它。

在初次安装后建立受保护的其他文件，既可以由你自己将其作为安装后配置过程的一部分来建立，也可以由服务器在运行时建立。由你自己建立的文件包括可选项文件或 SSL 相关文件。服务器在工作期间建立的目录和文件包括数据库目录、在这些目录下对应于数据库内数据表的文件、日志文件和 Unix 套接字文件。

显然，要保守由服务器维护的数据库的秘密。数据库所有者要经常和正确地考虑数据库内容的保密问题。即使他们不保密，也应由他们来选择是否使数据库内容公开，不会使其内容通过无足够保护的数据库目录泄露出去。

日志文件必须保密，因为它们包含客户发送到服务器的查询正文。这是个一般性问题，因为具有日志文件访问的任何人都可以监视数据库内容的改变。有关日志文件的更特殊的安全问题是其包括了敏感语句，包括口令。MySQL 利用口令加密，但这适用于口令已经设置后的连接建立。设置一个口令的过程包含查询，例如 `CREATE USER`、`GRANT` 或 `SET PASSWORD`，而且这些语句以简明的形式记录在某些日志中。读取访问日志的闯入者可能会发现敏感性信息，通过运行日志文件上 `grep` 这样简单的操作就可查找如 `GRANT` 或 `PASSWORD` 的口令。

客户程序一定可以访问其他文件，例如 Unix 套接字文件。但通常你要允许访问文件，而不允许控制它们。例如，用户应该能通过套接字文件连接至服务器，但他们不能删除套接字文件，否则就会损坏客户连接至本地服务器的能力。

13.1.1　如何偷取数据

下面提供了一个简短的例子来证明安全的重要性，着重说明不要其他用户直接访问 MySQL 数据目录的原因。

MySQL 服务器提供了一个灵活的权限机制，它是通过 `mysql` 数据库中的权限表实现的。你可以设置这些权限表的内容，以任何方式许可或拒绝客户对数据库的访问。这就保证了安全性，防止未经授权的网络访问你的数据。然而，假如在服务器上的其他用户可直接访问数据目录的内容，为了通过网络访问数据库而设置安全是没有价值的。除非你知道在 MySQL 服务器运行的机器上只有你一个人注册过，你得担心可能有人在那台机器上访问数据目录。

显然，你不想要服务器主机上的其他用户能直接写访问数据目录文件，因为他们可能会毁掉你的全部状态文件或数据库里的数据表。但是直接读访问同样有危险。如果数据表文件可以读，就意味着很容易地偷取该文件，并让 MySQL 向你展示出该数据表的内容。怎么样？如下所述。

(1) 在服务器主机上安装你自己的不合法的 MySQL 服务器，具有一个端口、套接字和数据目录，这些要和法定服务器所使用的不相同。

(2) 运行 `mysql_install_db`，初始化数据目录，你将可以作为 MySQL `root` 用户访问你的服务器，然后设置一个 `test` 数据库当做偷来的数据表的储藏室。

(3) 访问你要攻击的服务器数据目录，把对应于数据表的文件或你要偷取的数据表复制到你自己服务器数据目录下的 `test` 目录中。这个操作只需要读访问目标数据目录。

(4) 启动你的不合法服务器。多快！`test` 数据库现已装有偷来的数据表的副本，你可以随愿访问。`SHOW TABLES FROM test` 展示出你有哪个表的副本，`SELECT*`展示任何一个数据表的全部内容。

(5) 如果你真的想要胡来，对服务器上所有不具名用户账号开放许可权，使得任何人都能从任何主机连接至该服务器来访问你的 test 数据库。这就能有效地向全球公开偷来的数据表。

思考一下这种情况，然后回过来展望一下。难道你想要谁这样做吗？当然不是。所以应该用下面的指导原则来保护你自己。

13.1.2　保护你的MySQL安装

这里介绍如何设置组成 MySQL 安装的目录和文件的所有权和访问方式。这些指导原则用于对那些给定安装所有权的用户名和组名使用 mysql。（不管用户是谁，都不会是 root，原因参见 12.2.1 节的第 1 小节。）这些指导原则还认为在最初的布局下，MySQL 的所有安装部分位于同一个基本目录下，而不是分散在整个文件系统的不同位置上。安装的基本目录是/usr/local/mysql，而数据目录是在路径名/usr/local/mysql/data 之下。

完成这步之后，我们将叙述如何处理某些非标准的安装布局。系统布局可以随文中所述的各个方面而变化，但你应该能够采用合适的原则。按照需要为你自己的系统改变名称和路径名。如果你要运行多个服务器，应该对每一个服务器完成这种过程。

通过执行 ls -la 命令，你可以确定在你的数据目录中是否包含有不安全的文件或目录。查看具有“组”或“其他”已打开的权限的文件或目录。下面是不安全的数据目录清单，其中有一些数据库目录：

```
% ls -la /usr/local/mysql/data
total 10148
drwxrwxr-x   11 mysql  wheel     1024 May  8 12:20 .
drwxr-xr-x   22 root   wheel      512 May  8 13:31 ..
drwx------    2 mysql  mysql      512 Apr 16 15:57 menagerie
drwxrwxr-x    2 mysql  wheel      512 Jun 25  1998 mysql
drwx------    7 mysql  mysql     1024 May  7 10:45 sampdb
drwxrwxr-x    2 mysql  wheel     1536 Jun 25  1998 test
drwx------    2 mysql  mysql     1024 May  8 18:43 tmp
```

“.”代表列出的目录，也就是/usr/local/mysql/data。“..”代表父目录/usr/local/mysql。有些数据库目录有合适的许可权限：drwx------允许读、写和执行对所有者的访问，而不是访问任何人。其他目录对访问权限的要求很宽松：drwxrwxr-x 允许读和访问其他所有用户，甚至包括不是 mysqlgrp 组的用户。本例给出的状况是按时间得出的，从旧的 MySQL 安装开始，逐次升级到新版本。旧的 MySQL 服务器建立的权限限制较少，设定权限不如新服务器严格。（注意，更多限制的数据库目录，menagerie、sampdb 和 tmp 都具有最新的日期。）当前的 MySQL 服务器在数据库目录上设置权限，建立它是为了只让运行的账号访问。

你也可以使用 ls -1a 命令来检查 MySQL 安装的基目录/usr/local/mysql。例如，你可以得到一个如下所示的结果：

```
% ls -la /usr/local/mysql
total 44
drwxrwxr-x   13 mysql  mysql     1024 May  7 10:45 .
drwxr-xr-x   24 root   wheel     1024 May  1 12:54 ..
drwxr-xr-x    2 mysql  mysql     1024 Jul 16 20:58 bin
drwxrwxr-x   12 mysql  wheel     1024 May  8 12:20 data
drwxr-xr-x    3 mysql  mysql      512 May  7 10:45 include
drwxr-xr-x    2 mysql  mysql      512 May  7 10:45 info
```

```
drwxr-xr-x    3 mysql  mysql      512 May  7 10:45 lib
drwxr--r-x    2 mysql  mysql      512 Jul 16 20:58 libexec
drwxr-xr-x    3 mysql  mysql      512 May  7 10:45 man
drwxr-xr-x    6 mysql  mysql     1024 May  7 10:45 mysql-test
drwxr-xr-x    3 mysql  mysql      512 May  7 10:45 share
drwxr-xr-x    7 mysql  mysql     1024 May  7 10:45 sql-bench
```

data 目录的权限和所有权必须按照已有的指示进行改变。可能做的其他改变是限制访问 libexec 目录，其在 mysqld 服务器工作的地点。除了 MySQL 管理员，没有人需要访问服务器，所以可以使那个目录专用于 mysql 登录账户。

为了纠正刚才所述的问题，使用下列步骤。总的想法是锁住只有 mysql 能访问的任何内容，而其他用户需要访问（是合理的）的这些部分内容除外：

(1) 如果 MySQL 服务器正运行，则关停：

```
% mysqladmin -p -u root shutdown
```

(2) 使用下列命令把整个 MySQL 装置的所有者和组名分配设置在 MySQL 管理账号下，该命令必须以 root 执行：

```
# chown -R mysql /usr/local/mysql
# chgrp -R mysql /usr/local/mysql
```

另一种普通方法是让 root 拥有除了数据目录以外的一切，这可以按下面这样实现：

```
# chown -R root /usr/local/mysql
# chgrp -R mysql /usr/local/mysql
# chown -R mysql /usr/local/mysql/data
# chgrp -R mysql /usr/local/mysql/data
```

如果你设定总的所有权为 root，则必须以 root 执行下列大多数步骤，否则，你可以用 mysqladm 执行。

(3) 对于客户能访问的基目录及其任何子目录来讲，改变它们的方式，使得 mysql 有全部访问权限，而其他人只有读和执行的权限。这或许是它们早已设定好，如果没有，则改变它们。例如，基目录可以用下列任何一个命令设定：

```
% chmod 755 /usr/local/mysql
% chmod u=rwx,go=rx /usr/local/mysql
```

同样地，包括客户程序的 bin 目录可以用下列任一命令设定：

```
% chmod 755 /usr/local/mysql/bin
% chmod u=rwx,go=rx /usr/local/mysql/bin
```

(4) 客户程序不需要去访问的子目录可以设置成只允许 mysql 用户访问。用来存放 MySQL 服务器程序文件的 libexec 子目录就是一个例子。下面两条命令中的任何一条都可以把它的访问模式设置得更安全：

```
% chmod 700 /usr/local/mysql/libexec
% chmod u=rwx,go-rwx /usr/local/mysql/libexec
```

(5) 把数据目录以及它里面所有的文件和子目录的访问模式改变为只允许 mysql 用户访问：

```
% chmod -R go-rwx /usr/local/mysql/data
```

这样，除用来运行 MySQL 服务器的 mysql 账户外，任何其他的账户就都不能直接访问数据目录里的东西了。

完成前面的说明之后，MySQL 基本安装目录将有所有权和查看某些内容的权限，如下所示：

```
% ls -la /usr/local/mysql
total 44
drwxr-xr-x   13 mysql  mysql     1024 May  7 10:45 .
drwxr-xr-x   24 root   wheel     1024 May  1 12:54 ..
drwxr-xr-x    2 mysql  mysql     1024 Jul 16 20:58 bin
drwx------   12 mysql  mysql     1024 May  8 12:20 data
drwxr-xr-x    3 mysql  mysql      512 May  7 10:45 include
drwxr-xr-x    2 mysql  mysql      512 May  7 10:45 info
drwxr-xr-x    3 mysql  mysql      512 May  7 10:45 lib
drwx------    2 mysql  mysql      512 Jul 16 20:58 libexec
drwxr-xr-x    3 mysql  mysql      512 May  7 10:45 man
drwxr-xr-x    6 mysql  mysql     1024 May  7 10:45 mysql-test
drwxr-xr-x    3 mysql  mysql      512 May  7 10:45 share
drwxr-xr-x    7 mysql  mysql     1024 May  7 10:45 sql-bench
```

如上所示，所有东西现在都属于 mysql 用户组里的 mysql 用户了。清单中的第二项“..”代表着 /usr/local/mysql 子目录的父目录。这个父目录属于 root 用户，也只允许 root 用户修改。应该这样设置，因为你肯定不想让没有获得授权的用户把 MySQL 基本安装目录的父目录弄得一团糟。

在基目录下的数据目录的权限更严格：

```
% ls -la /usr/local/mysql/data
total 10148
drwx------   11 mysql  mysql     1024 May  8 12:20 .
drwxr-xr-x   22 mysql  mysql      512 May  8 13:31 ..
drwx------    2 mysql  mysql      512 Apr 16 15:57 menagerie
drwx------    2 mysql  mysql      512 Jun 25  1998 mysql
drwx------    7 mysql  mysql     1024 May  7 10:45 sampdb
drwx------    2 mysql  mysql     1536 Jun 25  1998 test
drwx------    2 mysql  mysql     1024 May  8 18:43 tmp
```

其中“..”行是指数据目录的父代目录，即是 MySQL 基本目录。

对于某些文件，不能采取“只有 mysql 能访问数据目录”的策略。例如，当你在数据目录中建立 Unix 套接字文件时，必须要打开一点来访问该目录，因为要把客户可选项放在该文件中。不然的话，客户程序不能通过套接字连接到)。如果要允许客户程序访问套接字文件，而又不能完全访问数据目录，可使用下列命令：

```
% chmod go+x /usr/local/mysql/data
```

为了避免这样打开数据目录。为 Unix 套接字文件使用一个不同的位置，如基目录。这种原则同样适用于除 mysql 以外的程序需要正常访问的其他文件，例如包含全局客户参数的选项文件。

如前所述，前面的步骤认为和 MySQL 有关的所有文件放置在一个基目录下。如果不是这样，必须定位每个和 MySQL 有关的目录，并针对每一个执行相应的操作。例如，如果数据目录位于 /var/mysql/data，而不是在/usr/local/mysql 下面，则必须发出几个命令正确地改变你装置的所有权：

```
# chown -R mysql /usr/local/mysql
# chgrp -R mysql /usr/local/mysql
# chown -R mysql /var/mysql/data
# chgrp -R mysql /var/mysql/data
```

假定你在 MySQL 装置目录下建立了一个 innodb 目录，在该目录下保存和 InnoDB 有关的所有文件。这些文件默认地放置在数据目录中。如果你把它们放在 innodb 目录中，则必须设置该目录具有和数据目录同样的访问模式。当重定位平常放在数据目录中的其他文件时，这个原则同样适用，例如重定位日志文件。

如果在安装 root 下某些目录实际上是指向另一地方的符号链接，这会产生其他困难。如果 chown 和 chgrp 的版本没有随从符号链接，则必须跟踪它，并在链接指向的位置改变所有权。做这项工作的一种方法是使用 find：

```
# find /usr/local/mysql -follow -print | xargs chown mysql
# find /usr/local/mysql -follow -print | xargs chgrp mysql
```

改变访问模式也要考虑这些。例如，如果在你数据目录下存在符号链，而且 chmod 不跟从它们，则使用下面命令替代：

```
# find /usr/local/mysql/data -follow -print | xargs chmod go-rwx
```

因为该点数据目录内容的所有权和权限设定成只允许 mysql 用户访问，应该保证从现在起服务器以 mysql 运行，较容易的方法是指定/etc/my.cnf 文件或服务器启动时读取的其他 my.cnf 文件的 [mysqld]组件：

```
[mysqld]
user=mysql
```

用这种方法，服务器以 mysql 运行，不管你启动服务器时是以 root 注册，还是以 mysql 注册。使用特定注册账号运行服务器的其他信息已在 12.2.1 节的第 1 小节讨论过。

保护 MySQL 的安装后，则可以重新启动服务器。

1. 保护套接字文件

服务器使用 Unix 区域套接字文件使客户连接至 localhost。套接字文件通常可公开访问，以便客户程序使用它。然而，不能放在有删除权限的任意客户的目录中。例如，通常是把套接字文件建立在/tmp 目录中，但是在某些 Unix 系统上，该目录具有允许用户删除自己以外的文件的权限。这意味着任何用户可以去除套接字文件。因此，要防止客户程序建立与用户的 localhost 连接，除非服务器重新启动再建立套接字文件。最好是/tmp 目录具有“粘滞位”设定，即使有人能在该目录中建立文件，用户也只能删除自己的文件。可以通过执行以下命令来为目录设置粘滞位。

```
# chmod +t /tmp
```

某些装置把套接字文件放在数据目录中，如果使数据目录专用于 mysql:，则会产生一个问题：客户不能访问套接字文件，除非是以 root 或 mysql 运行。在这种情况下，一个可选项可稍微打开数据目录，以便客户可以看到套接字文件：

```
% chmod go+x /usr/local/mysql/data
```

不然的话，可以改变服务器建立套接字文件的位置。例如，可以配置 MySQL，通过指定/usr/local/mysql/mysql.sock 的一个位置来在基目录中创建文件。既可以在全局选项文件中指定位置，也可以由源重新编译建立在不同的默认位置中。如果你选择使用一个可选项文件，一定要为服务器和客户都指定该位置。

```
[mysqld]
socket=/usr/local/mysql/mysql.sock
```

```
[client]
socket=/usr/local/mysql/mysql.sock
```

重新编译的工作更多，但这是更圆满的解决办法，因为使用选项文件对不检查可选项文件的客户不起作用。（所有标准的 MySQL 客户能行，但第三程序不行。）通过重新编译，新的套接字位置将变成客户库知道的默认值。这样的话，任何使用客户库的程序将得到新的位置作为自己的默认值，而不管其是否使用选项文件。

2. 保护选项文件

选项文件是潜在的信息泄露源，它们所包含的选项不应该被泄露。

- 如果某个选项文件包含着诸如MySQL账户名或口令之类的敏感信息，就不应该让外界的人们有机会访问到它。
- 对/etc/my.cnf文件的访问通常是不受限的，因为它是设定全局级客户选项的常见位置。这意味着你不应该把服务器级的选项如复制机制的口令等保存在它里面。
- 每个用户专用的.my.cnf选项文件应该只出现在该用户的主目录里，并且应该只允许该用户拥有和访问。如果想保护这个专用文件，请在你自己的主目录里执行以下命令：

```
% chmod u=rw,go-rwx .my.cnf
```

- 其他选项文件需要根据它们的具体用途来设置它们的访问模式。

有个办法可以确保用户专用选项文件有安全的访问模式和所有权设置：运行一个程序依次检查每位用户主目录里的.my.cnf 文件，只要发现安全隐患就加以纠正。如下所示的 Perl 脚本 chk_mysql_opt_files.pl 可以完成这个任务：

```
#!/usr/bin/perl
# chk_mysql_opt_files.pl - check user-specific .my.cnf files and make sure
# the ownership and mode is correct. Each file should be owned by the
# user in whose home directory the file is found. The mode should
# have the "group" and "other" permissions turned off.

# This script must be run as root.  Execute it with your password file as
# input.  If you have an /etc/passwd file, run it like this:
#  chk_mysql_opt_file.pl /etc/passwd
# For Mac OS X, use the netinfo database:
#  nidump passwd . | chk_mysql_opt_file.pl

use strict;
use warnings;

while (<>)
{
  my ($uid, $home) = (split (/:/, $_))[2,5];
  my $cnf_file = "$home/.my.cnf";
  next unless -f $cnf_file;      # is there a .my.cnf file?
  if ((stat ($cnf_file))[4] != $uid)  # test ownership
  {
    warn "Changing ownership of $cnf_file to $uid\n";
    chown ($uid, (stat ($cnf_file))[5], $cnf_file);
  }
  my $mode = (stat ($cnf_file))[2];
```

```
  if ($mode & 077)              # test group/other access bits
  {
    warn sprintf ("Changing mode of %s from %o to %o\n",
                  $cnf_file, $mode, $mode & ~077);
    chmod ($mode & ~077, $cnf_file);
  }
}
```

你可以在 sampdb 分配的 admin 目录中找到 chk_mysql_opt_files.pl。必须以 root 运行这个脚本程序，因为它应能改变其他用户所拥有文件的方式和所有权。为了自动执行该脚本程序，把它设置成由 root 运行的一个 cron 作业。

13.2　外部安全性：防止未经授权的网络访问

MySQL 的安全系统是灵活的。允许你以许多不同方法设置用户访问权限。通常，使用 CREATE USER、GRANT 和 REVOKE 等账户管理语句做这项工作，它们为你修改控制客户访问的权限表。然而，你可能发现用户权限似乎不按你想的方式工作。对于这种情况，去了解 MySQL 权限表的结构和服务器如何使用它们来确定访问权限是有帮助的。了解这些后，你可以直接修改权限表来增加、去除或修改用户权限，你可以在检查这些表格时诊断权限的问题。

这里假设你已阅读了 12.4 节，并且了解各种账户管理语句如何工作。这些语句提供了一种方便的方法让你设置用户账号和与其相关的权限，但它们只不过是一个前端。所有实际操作是在 MySQL 权限表中进行的。

13.2.1　MySQL权限表的结构和内容

通过网络连接至服务器的客户访问 MySQL 数据库时要受权限表内容的控制。这些表放在 mysql 数据库中，并在 MySQL 安装至机器上的过程中第一次初始化（附录 A 将叙述）。这些表格名叫 user、db、tables_priv、columns_priv 和 Procs_priv。它们的使用如下所述。

- user表列出可以连接至服务器的用户账号、用户口令以及每个用户有的全局(超级用户)权限（如果有的话）。重要的是要知道，在user表中启用的任何权限是适用于所有数据库的全局性权限。例如，如果你启用了user表登记项中的DELETE权限，和该登记项相关的账号能从任何数据库中的任一表中删除记录。在你这么做时要认真考虑。

  因为超级用户的权限性质是在 user 表中指定的，通常最好使该表中登记项用的所有权限保持断开，而列出其他表中限制更高的权限。对这种原则有两种例外。

  首先，超级用户，例如 root 和其他管理账号，需要全局性权限来操作服务器。这些账号往往很少。

  其次，一些专用全局性权限通常可以可靠地授予。这些涉及建立临时数据表，锁止数据表，并能使用 SHOW DATABASES 语句。大多数安装大概会授予这些，但控制较紧的其他安装一定不会。

  user 表亦有 SSL 选项用的列，和与 SSL 建立安全连接有关，还有资源管理用的列，可以防止给定的账号独占该服务器。
- db表列出了哪个账号有权限使用哪个数据库。如果在其中授予权限，则用于数据库内所有数据表、存储例程等。

- tables_priv表指定数据表级的权限。其中指定的权限用于数据表中所有列。
- columns_priv表指定数据列级的权限。其中指定的权限用于数据表中的特定数据列。
- procs-priv数据表记录着各种存储例程（存储函数和存储过程）的权限。在这里设定的权限将作用于数据库里的某个特定的例程。这个数据表是从MySQL 5.0.3版开始新增的。

mysql 数据库还包含一个名为 host 的权限数据表，该数据表要和 db 数据表配合使用。不过，host 数据表已经过时了，所以本书不再对它多做讨论。

表 13-1 至表 13-4 对每个权限数据表的结构进行了剖析，有关信息按数据列的类型分类归纳。每一个权限数据表都包含有两种基本类型的数据列：其一是作用范围数据列，用来确定权限数据表里的某个数据行应该在什么场合生效；其二是权限数据列，用来确定某个数据行具体授予的是什么权限。针对系统管理性操作和针对某种特定数据库对象的操作，权限数据列又可以进一步细分为几个类别。user 数据表还有几个针对 SSL 连接和资源管理操作的数据列，它们是 user 数据表独有的，其作用范围是系统全局。有些权限数据表还包含其他一些无法纳入上述分类的数据列，但因为它们与账户管理工作无关，这里不多说它们了。

表 13-1 权限表的访问范围列

user表	db表	tables_priv表	columns_priv表	procs_priv表
Host	Host	Host	Host	Host
User	User	User	User	User
Password	Db	Db	Db	Db
		Table_name	Table_name	Routine_name
			Column_name	Routine_type

表 13-2 权限表的管理权限列

user表	db表	host表
Create_user_priv		
File_priv		
Grant_priv	Grant_priv	Grant_priv
Process_priv		
Reload_priv		
Repl_client_priv		
Repl_slave_priv		
Show_db_priv		
Shutdown_priv		
Super_priv		

表 13-3 权限表对象权限列

user表	db表
Alter_priv	Alter_priv
Alter_routine_priv	Alter_routine_priv
Create_priv	Create_priv
Create_routine_priv	Create_routine_priv
Create_tmp_table_priv	Create_tmp_table_priv
Create_view_priv	Create_view_priv
Delete_priv	Delete_priv
Drop_priv	Drop_priv

（续）

user表	db表	
Event_priv	Event_priv	
Execute_priv	Execute_priv	
Index_priv	Index_priv	
Insert_priv	Insert_priv	
Lock_tables_priv	Lock_tables_priv	
References_priv	References_priv	
Select_priv	Select_priv	
Show_view_priv	Show_view_priv	
Trigger_priv	Trigger_priv	
Update_priv	Update_priv	
tables_priv表	**columns_priv表**	**procs_priv表**
Table_priv	Column_priv	Proc_priv
Column_priv		

表 13-4　权限表 SSL 和资源管理列（仅 **user** 表可用）

SSL列	资源管理列
ssl_type	max_connections
ssl_cipher	max_questions
x509_issuer	max_updates
x509_subject	max_user_connections

权限表系统包括 tables_priv、columns_priv 和 procs_priv 表，用于定义特定的数据表、列、存储函数和过程的权限。然而，没有 rows_priv 表，因为 MySQL 不提供行级的权限。例如，你不能限制用户只访问表中的这些行，该表中还包括某些列中的特定值。如果你必须这样做，那得使用应用程序提供。实现行级锁定的一种方法是使用 GET-LOCK()和 RELEASE_LOCK()函数执行咨询锁定，参见 C.2.8 节。

MySQL 新版本有时添加新权限。例如，Event_priv 和 Trigger_priv 列在 MySQL 5.1.6 中得到了实现。如果要把现有的 MySQL 升级到这样的高版本，就需要在使用新权限之前升级权限表。12.3 节描述了这一过程。

1. 权限表的访问范围列

权限表范围列用于在给定账号试图执行一个给定操作时确定使用哪些行。每个权限表的行包括 Host 和 User 列，指示该行适用于特定用户从给定主机进行连接。例如，在 Host 和 User 列中具有 localhost 和 bill 的用户表记录可用于 bill 从本地主机进行连接，而不是用于 betty 的连接。其他表包括其他范围列。db 表包括一个 db 列，指出该行适用于哪个数据库。同样地，tables_priv 和 columns_priv 表中各行包括了范围列，进一步使其范围变窄成数据库中一个特定表或一个表中的列。procs_priv 范围列指定某行应用于哪个存储函数或过程。

2. 权限表的权限列

权限表也包括权限列。对于每一行这些列指出和范围列所列出值相匹配的用户具有什么样权限。MySQL 支持的权限在下面示出，其有管理权限和控制数据库和数据表访问的权限。每一列表使用 GRANT 语句用的权限名。通常，这些权限名具有和权限表中权限列名称显然相似的名。例如，SELECT 权限对应于 Select_priv 列。

3. 管理权限

下列权限适用于控制服务器运行的管理操作，或是用户授予权限的能力。

- CREATE USER。你可以使用CREATE USER、DROP USER、RENAME USER和REVOKE ALL PRIVILEGES语句。MySQL 5.0.3引入了这一权限。
- FILE。用于让服务器在服务器主机上读或写文件。为了保持在一定界限内使用这个权限，服务器一定要采取些预防措施。
  - 只可以访问全球可读的文件，这样就不用考虑以什么方法来保护。
  - 你要写的文件不能已经存在。这可防止你强迫服务器覆盖重要文件，例如/etc/passwd 或属于别人数据库内的数据库文件。（如果不实施这种限制，你完全可能更换 mysql 数据库中权限表的内容。）

  尽管有这些预防措施，没有合理的理由不应授予这个权限。因为这可能有极大的危险性，13.2.4节将会讨论。如果你授予 FILE 权限，一定不要以 Unix root 用户运行服务器，因为 root 可以在文件系统任何地方建立新文件。只要由普通注册账号运行服务器，则服务器只能在那个账号可以访问的目录中建立文件。参见 12.2.1 节的第 1 小节。
- GRANT OPTION。你可以向其他用户授予你自己的权限，其中也包括GRANT OPTION权限。
- PROCESS。MySQL服务器是多线程的，这样的话，它可以同时服务多个客户连接。这些线程可以认为是在服务器内运行的进程。在MySQL4.0.2之前，PROCESS权限允许你使用SHOW PROCESSLIST语句或mysqladmin processlist命令来观察正在执行的线程信息。只要加了这个权限，即使这些线程和其他用户有关，也能查看所有线程。即便没有PROCESS权限，你也可以查看自己的活动。
- RELOAD。你可以执行各种服务器管理操作。利用该权限可以发出FLUSH和RESET等语句。还可以执行下列mysqladmin命令：reload、refresh、flush-hosts、flush-logs、flush-privileges、flush-status、flush-tables和flush-threads。
- REPLICATION CLIENT。使用SHOW MASTER STATUS和SHOW SLAVE STATUS查询主服务器和从服务器的地点。
- REPLICATION SLAVE。客户可连接至主服务器，并请求从服务器修改，以及使用SHOW MASTER STATUS和SHOW SLAVE STATUS。这一权限必须授给用来连接主服务器的从服务器账户。
- SHOW DATABASES。让你有权通过SHOW DATABASES语句列出所有数据库的名字。如果不具备这个权限，就只能列出你有权使用的数据库的名字。不过，列出所有数据库名字的能力可以通过任何一种作用于数据库的全局级权限赋给用户，就连CREATE TEMPORARY TABLES和LOCK TABLES权限也不例外，而这两种权限往往会是全局级的。如果想确保只有具备SHOW DATABASES权限的用户才能使用SHOW DATABASES语句的话，请使用--skip-show-database选项来启动MySQL服务器。
- SHUTDOWN。让你有权使用mysqladmin shutdown命令或通过其他手段关停MySQL服务器。
- SUPER。让你有权使用KILL语句或mysqladmin kill命令“杀死”服务器进程。事实上，这个权限让你有能力结束任何进程，甚至是那些与其他用户相关联的进程。无论你是否具备SUPER权限，总是可以结束自己的进程。

  这个权限还能让你有权使用 SET 语句去修改全局级系统变量和全局级事务隔离级别，有权使用 CHANGE MASTER、PURGE MASTER LOGS、SHOW MASTER STATUS、SHOW SLAVE STATUS、

START SLAVE 和 STOP SLAVE 等语句。SUPER 权限还能让你有权使用 DES_DECRYPT()函数和保存在 DES 密钥文件中的密钥进行 DES 解密操作。

SUPER 权限还能让你有权使用 mysqladmin debug 命令，并在连接服务器时重写 max_connections 设置。所以就算分配给普通用户的所有连接通道都已被占用，你也可以通过 MySQL 服务器为系统管理性连接而保留的连接通道去访问它。在 MySQL 5.1.6 之前的版本里，SUPER 权限还能让你有权添加或丢弃触发器；在那之后的版本里，需要具备 TRIGGER 权限才能执行那些操作。

4. 数据库和数据表权限

下列权限适用于数据库和数据表上的操作。

- ALTER。可使用ALTER TABLE语句，虽然你或许还需要其他权限，这取决于你想要该表做什么。
- ALTER ROUTINE。可修改或删除存储函数和过程。MySQL 5.0.3引入了这一权限。
- CREATE。可建立数据库和表。该权限不允许你在数据表上建立索引，除非这些已在CREATE TABLE语句中说明过。
- CREATE ROUTINE。可以创建存储函数和过程。MySQL 5.0.3引入了这一权限。
- CREATE TEMPORARY TABLES。可以利用CREATE TEMPORARY TABLE语句创建临时表。
- CREATE VIEW。可创建视图。MySQL 5.0.1 引入了这一权限。
- DELETE。可从数据表中去除现有的记录。
- DROP。删除数据库和数据表，该权限不允许删除索引。
- EVENT。让你有权操控事件调度程序去管理各种事件。这个权限是在MySQL 5.1.6版本里引入的。
- EXECUTE。让你有权执行存储函数和存储过程。这个权限是在MySQL 5.0.3版本里引入的。（它在以前的版本里也存在，但没有任何实际用途。）
- INDEX。让你有权为数据表创建或丢弃索引、为索引分配键缓存、把索引预加载到键缓存，等等。
- INSERT。让你有权把新数据行插入到数据表。
- LOCK TABLES。让你有权通过发出LOCK TABLES语句来锁定数据表。这个权限只对你还拥有SELECT权限的数据表起作用，但可以让你设置读操作锁或写操作锁，而不是只能设置读操作锁。这个权限对于因为语句执行过程中隐含的要求而由MySQL服务器替你施用的锁没有效力。那些锁是自动启用和释放的，与你的LOCK TABLES权限设置没有关系。
- REFERENCES。这个权限目前还没有实际使用。它今后可能会被用来定义谁有权制定外键约束条件。
- SELECT。让你有权使用SELECT语句从数据表检索数据。对于诸如SELECT NOW()或者SELECT 4/2之类只对表达式进行求值、不涉及任何数据表的SELECT语句来说，这个权限不是必不可少的。
- SHOW VIEW。让你有权使用SHOW CREATE VIEW语句去查看视图的定义。这个权限是在MySQL 5.0.1版本里引入的。
- TRIGGER。让你有权添加或丢弃触发器。这个权限是在MySQL 5.1.6版本里引入的。在那之前，执行与触发器有关的操作需要具备SUPER权限。
- UPDATE。让你有权对数据表里的现有数据行进行修改。

有些操作需要同时具备多种权限才能执行。比如说，REPLACE 操作其实是先 DELETE 再 INSERT，所以需要同时具备 DELETE 和 INSERT 权限才能执行。

5. 授权表如何表示权限

在 user 和 db 表中，每个权限以逗号分隔。这些列全部定义为有一种 ENUM（'N'，'Y'），默认值为'N'（off）。例如 select_priv 列定义如下：

```
Select_priv ENUM('N','Y') CHARACTER SET utf8 NOT NULL DEFAULT 'N'
```

tables_priv 和 columns_priv 表中的权限用 SET 表示，其允许储存在一列中的权限任意组合。tables_priv 表中的 Table_priv 列定义如下（MySQL 5.1.6 之前没有 Trigger）：

```
SET('Select','Insert','Update','Delete','Create','Drop','Grant',
    'References','Index','Alter','Create_view','Show_view','Trigger')
CHARACTER SET utf8 NOT NULL DEFAULT ''
```

columns_priv 表中的 Column_priv 列定义如下：

```
SET('Select','Insert','Update','References')
CHARACTER SET utf8 NOT NULL DEFAULT ''
```

列权限比表权限少的原因是在列一级有意义的操作较少。例如，你可以从要去除的表中删除一行，但你不能删除一行中各个列。

注意，INSERT 存在于列级别。如果只是表中某些列有 INSERT 权限，那么你可以在插入新行时只为那些列指定值，其他列将被插入默认值。

tables_priv、columns_priv 和 procs_priv 表比 user 和 db 新，所以它们用更有效的 SET 表示来列出一列中的多个权限。

用户表具有几个管理权限列，不在其他任何权限表中出现，例如 File_priv、process_priv、reload_priv 和 Shutdown_priv。这样的权限仅仅在 user 表中出现，因为它们是全局权限，和任何特定数据库或表无关。例如，根据目前默认数据库是什么来允许或不允许用户关停服务器是没有意义的。

6. 权限表的SSL相关列

用户表中的一些列适用于鉴别 SSL 上的安全连接。主要的列是 ssl_type，其指明是否要安全连接和需要什么类型的安全连接。它以 4 个可能值的 ENUM 表示。

```
ENUM('','ANY','X509','SPECIFIED') CHARACTER SET utf8 NOT NULL DEFAULT ''
```

ssl_type 枚举值有以下含义。

- ' '表示不需要安全连接，这是个默认值，当你设置一个账号，而不是指定一个REQUIRE子句时使用，或者当你明确指定REQUIRE NONE时使用。
- 'ANY'表示必须安全连接，而且可以是任何一种安全连接，这是一种“类属”需要。当你在GRANT语句中指定REQUIRE SSL时，该列设置成这个值。
- 'X509'表示需要安全连接，但是客户必须提供一个有效的X509证书。证书的内容和其他无关。当你指定REQUIRE X509时，该列设成这个值。
- 'SPECIFIED'表示安全连接必须满足专门的要求。当你在REQUIRE子句中指定ISSUER、SUBJECT或CIPHER值时，该列设定成这个值。

对于'SPECIFIED'以外的所有 ssl_type 值，当确定客户连接意图时，服务器忽略在其他 SSL 相关列中的数值。对于'SPECIFIED'来说，服务器检查其他列，对于不空的任何值，客户必须提供相匹配的信息。

- ssl_cipher。如果非空，该列表示在连接时客户必定要使用密码方法。这可以防止客户使用不好的密码方法。
- x509_issuer。如果非空，该列表示在客户提供的X509证书中一定可找到发出者的值。

- x509_subject。如果非空，该列表示在客户提供的X509证书中一定可找到题目值。

ssl_cipher、x509_issuer 和 x509_subject 都是在 user 表中以 BLOB 列表示。

有关使用 SSL 进行安全连接的其他信息，参见 13.3 节。

7. 权限表的资源管理列

user 表中的下面各列可限定任一给定 MySQL 账号所消耗服务器资源的范围。

- max_connections是允许某给定账户在一小时内连接到服务器的最大次数。该值为零意味着“没有限制”。虽然这个数据列与max_connections系统变量同名，但两者并没有内在联系。
- max_questions是允许某给定账户在一小时内发出的语句的最大个数。该值为零意味着“没有限制”。
- max_updates的作用类似于max_questions，但更加具体，它针对的是数据修改类语句。该值为零意味着“没有限制”。
- max_user_connections是允许某给定账户同时保有的客户连接的最大个数。如果该值为零，MySQL服务器将根据全局级max_user_connections系统变量的值去确定一个上限值。如果该值大于零，其值将优先于max_user_connections系统变量。这个数据列是在MySQL 5.0.3版里引入的。

如果 MySQL 服务器重新启动，当前计数器将全部重置为零。在你重新加载权限数据表或是发出 FLUSH USER_RESOURCES 语句的时候，也会发生类似的重置，只有 max_user_connections 值不受影响。

关于设置账户限制的更多信息，请参阅 12.3.1 节的第 5 小节。

13.2.2　服务器如何控制客户访问

当你使用 MySQL 时，分二个阶段的客户访问控制。第一阶段是当你试图连接至服务器时。服务器探望 user 表查看能否找到一行和你连接的主机、名字以及提供的口令匹配。如果不匹配，就不能连接。如果匹配，该服务器仍要检查 user 表 SSL 和资源管理列：

- 如果超过了每小时连接数或同步链接数的极限值，则拒绝连接。
- 如果user表行指出需要安全连接，服务器确定你提供的证件是否和SSL相关列中需要的相匹配。如果不相配，则拒绝连接。

如果每个检查都通过了，服务器建立连接，进入第二阶段。对于安全连接来说，客户程序和服务器要使用加密。在第二阶段，MySQL 服务器将为你发出的每一条语句进行两项检查。首先，它检查“每小时语句数”和“每小时更新操作数”上限。然后，服务器检查权限数据表，以确认你有足够的访问权限去执行那条语句。先检查那两个上限值的原因是：如果你已经达到了这些上限，就没有必要再去检查你的权限了。第二阶段将一直持续到你与 MySQL 服务器断开连接为止。

下面的讨论更详细地论述了 MySQL 服务器的使用规则，以便权限表行和客户进入的连接请求以及查询相匹配。这包括了权限表范围列中合法值的类型，来自不同权限表的权限值如何组合，以及寻找给定权限表内行的次序。

1. 范围列的内容

每一个范围列由规则来控制，其规定哪一类数值是合法的，以及服务器如何来解读这些值。某些范围列需要文字值，但大多数允许通配符或其他专用值。

- Host。Host数据列的值可以是一个主机名或者是一个IP地址。代表本地主机的localhost值，用来匹配某给定客户从本地主机连接到服务器本地网络接口的情况，这些本地网络接口被定义为：

■ Unix 系统上的 Unix 套接字文件；
■ Windows 系统上的命名管道或共享内存；
■ TCP 回跳接口，即 IP 地址是 127.0.0.0 的那个接口。这适用于任何一种系统。

如果你使用主机的实际名或 IP 号连接时，则 localhost 是不匹配的。假定本地主机名是 cobra.snake.net，并且有个名叫 bob 的用户在 user 表中有两行记录，一个具有 localhost 的 Host 值，另一个具有 cobra.snake.net 的值。如果 bob 在 Unix 或 Windows 上以下列任一方式连接，则有 localhost 的值是匹配的：

```
% mysql -p -u bob -h localhost
% mysql -p -u bob -h 127.0.0.1
```

此外，在 Windows 上，如果 bob 如下连接，则 localhost 行是匹配的：

```
C:\> mysql -p -u bob -h .
C:\> mysql -p -u bob --protocol=pipe
C:\> mysql -p -u bob --protocol=memory
```

如果 bob 使用服务器主机名（cobra.snake.net）或与主机名对应的 IP 号从本地主机连接，具有 cobra.snake.net 的 Host 值的行是匹配的。在这两种情况下，连接将使用 TCP/IP。

你也可以使用通配符指定 Host 值。可以使用“%”和“_”SQL 模式字符，它们的意义和你在查询中使用 LIKE 操作符一样。（不允许用于 REGEXP 的正则表达式类型。）SQL 模式字符既可用于名字，也可用于 IP 号。例如，%.example.com 和 example.com 区域中的任何主机相匹配，而%.edu 和任何教学机构的任一主机相匹配。同样，10.0.%和 10.0 类别 B 子网络中的任一主机相匹配，192.168..3.%和 192.168.3 B 子网络相匹配。

%的 Host 值完全和任一主机匹配，并允许用户从任一地方连接。权限表中的空白 Host 值和%一样。有一个例外，权限表中的空白 Host 值表示“检查 host 表中更多的信息”。host 表过时了，所以不应该使用空白 Host 值。

你还可以在 Host 数据列给出一个带子网掩码的网络地址，这要求客户主机的 IP 地址必须与网络地址相匹配。比如说，192.168.128.0/255.255.255.0 给出了一个 24 位的网络地址，凡是 IP 地址的前 24 位等于 192.168.128 的客户主机都能与之匹配。你可以把这种 Host 数据列值想象成一种通配符。子网掩码的取值只能是 255.0.0.0、255.255.0.0、255.255.255.0 或 255.255.255.255。换句话说，总共 32 位的子网掩码的前 8、16、24 或 32 位必须全部是 1，其余的则必须是 0。

❑ User。用户名必须是文字值或空白。空白值和任何名字匹配，这意味着“不具名”。不然的话，该值精确地和指定的名字相匹配。作为User值的%的用户不表示空白，而是和具有文件名%的用户相匹配，这或许并不是你想要的。

当对照 user 表校验进入的连接时，如果第一个匹配的行具有一个空白的 User 值，则该客户被视为一个不具名用户。

❑ Password。Password值是空白或不空白，而且不允许有通配符。空白口令不意味着任何口令都匹配，这意味着用户一定没有指定口令。口令作为加密值存储，不是文本。如果你把一个文字口令存在Password列，则用户将不能连接！CREATE USER、GRANT语句和mysqladmin password命令给你自动加密，但是，如果你使用如INSERT、REPLACE、UPDATE或SET PASSWORD语句来直接修改权限表，一定要使用PASSWORD('new_password')而不是仅仅'new_password'来指定口令。

- Db。在db表中，Db值可以用文字指定，也可以使用“%”或“_”SQL模式字符来指定一个通配符。%或空白值和任一数据库相匹配。在columns_priv、tables_priv和Procs-priv表中，Db值必定是文本数据库名，而且和指定名精确匹配，不允许模式和空白值。
- Table_name、Column_name和Routine_name。这些列中的数值必定分别是文本表名、存储例程名或列名，而且和指定名精确匹配，不允许模式和空白值。
- Routine_type。这个数据列里的值只能是'FUNCTION'或'PROCEDURE'之一，该值决定着同一数据行的Routine_name数据列里的名字是作用于一个存储函数，还是作用于一个存储过程。Routine_name值和Routine_type值的组合独一无二地代表着由Db数据列的值所给定的那个数据库里的一个存储例程。

服务器对某些范围列区分大小写，然而其他则不是，如表 13-5 所示。注意，特别是 Db 和 Table_name 值必定区分大小写，即使 SQL 语句中数据库和数据表名是根据服务器运行的文件系统是否区分大小写来处理。（通常在 Unix 下区分大小写，而在 Windows 下不区分大小写。）

表 13-5　权限表范围列中的大小写情况

列	是否区分大小写
Host	否
User	是
Password	是
Db	是
Table_name	是
Column_name	否
Routine_name	否

口令是如何存储在用户表中

MySQL 服务器是在将口令存储到用户表之前用 PASSWORD()函数为口令加密的，以防止口令按普通文本泄露给具有读访该表的用户。人们公认，PASSWORD()和 Unix 口令所用的是同样一种加密，但事实并非如此。两种加密的相同点是单向和不可逆，但是 MySQL 使用的加密算法与 Unix 不一样。这意味着你即使使用 Unix 口令作为 MySQL 的口令，也不要指望该加密口令字符串相匹配。如果想要在一个 MySQL 用程序中执行 Unix 加密，则使用 CRYPT()函数，而不是 PASSWORD()。（如果你很想知道其他可用于你的应用程序的加密可选项，请参阅 C.2.7 节。）

2. 语句访问权限的验证

每当你发出一条 SQL 语句，服务器就会去检查你是否已经达到了你的语句资源上限。这些上限值是由 user 数据表里的 max_questions 和 max_updates 值设定的。如果还没有达到你的上限值，MySQL 服务器将检查你是否具备足够的访问权限去执行那条语句。服务器判断你是否具备足够权限的办法是核查 user、db、tables_priv、columns_priv 和 procs_priv 数据表里与你有关的权限设置。这种核查的结果无非两种，其一是 MySQL 服务器确定你有足够的权限；其二是它在搜索了所有上述数据表之后无功而返。具体流程如下所示。

(1) 服务器首先检查 user 数据表里与你的 MySQL 用户账户相匹配的那个数据行，看你都具备哪些全局级权限。如果你具备的全局级权限足以执行那条语句，MySQL 服务器才会执行。

(2) 如果你的全局级权限不够，服务器将到 db 数据表里寻找与你相匹配的数据行，如果找到了，它会把该数据行里的权限叠加到你的全局级权限上。如果叠加结果是你的权限足以执行那条语句，服务器将执行。

(3) 如果你的全局级权限和数据库级权限叠加在一起还不够，MySQL 服务器将检查 tables_priv、columns_priv 和 procs_priv 数据表，以确定你是否具备执行那条语句所需要的权限。

(4) 如果服务器在检查完所有上述数据表之后，发现你仍不具备执行那条语句的权限，它将拒绝执行该语句。

服务器根据权限数据表来判断你是否具备足够权限的过程，可以归结为一个如下所示的布尔表达式：

```
user OR db OR tables_priv OR columns_priv or procs_priv
```

以上描述无疑会让人们认为核查访问权限是一个相当复杂的过程，尤其是在想到服务器会核查每位客户发出的每条语句的时候。其实这个过程相当迅速，因为服务器为执行每条语句而检索的权限信息并非来自硬盘上的权限数据表。它会在启动时把那些数据表的内容读入内存，然后使用内存里的副本进行权限核查。这种安排给访问权限核查操作带来了显著的性能改善。不仅如此，如果你事先对权限设置进行过优化的话，就可以保证访问权限核查操作更有效率。在服务器把权限数据表读入内存的时候，它会留意是否存在有着资源限制的账户，是否存在有着数据表级、数据列级或例程级权限的账户。如果没有，它在对客户发出的语句进行访问权限核查的时候就不会去检索那些用不着的信息了。这意味着服务器可以省略访问权限核查全过程中的某些特定步骤。

使用权限数据表在内存里的副本进行访问权限核查有一个特别值得注意的负面效果：如果你直接修改了权限数据表的内容，MySQL 将不会注意到有关权限发生了改变。比如说，如果你通过一条 INSERT 语句向 user 数据表插入一个新数据行来增加一个新 MySQL 用户，那个新数据行并不能保证那个新用户一定可以连接到 MySQL 服务器。这类问题往往会让一部分系统管理员新手（甚至某些老手）困惑不已，但解决方案相当简单：在直接修改权限数据表之后让 MySQL 服务器重新加载它们的内容即可。你可以通过执行一条 FLUSH PRIVILEGES 语句或是执行 mysqladmin flushprivileges 或 mysqladmin reload 命令来做这件事情。

如果你是使用 CREATE USER、DROP USER、RENAME USER、GRANT、REVOKE 或 SET PASSWORD 语句去设置或修改 user 数据表的，就没有必要让 MySQL 服务器重新加载权限数据表了。服务器会把这些语句映射到相应的权限数据表修改操作上，然后自动更新那些数据表在内存里的副本。

3. 作用范围数据列的匹配顺序

MySQL 服务器会按照某种特定的顺序对权限数据表里的数据行进行排序，然后按顺序检查那些数据行能不能与外来连接相匹配。如此得到的第一个匹配决定了将使用哪一个数据行。因此，了解 MySQL 所使用的排序方法非常重要，尤其是用于 user 数据表的排序。有许多人就是从这个问题开始逐步深入了解 MySQL 系统安防领域的。

在读取 user 数据表的内容时，MySQL 服务器会按照 Host 和 User 数据列的值对数据行进行排序。排序时以 Host 数据列为主，所以 Host 值相同的数据行将集中排列在一起，它们之间的先后再按 User 数据列的值进一步排序。请注意，这类排序并不按字母表顺序进行，或者更准确地说，只在一定程度上如此。这里有个大家应该记住的原则：MySQL 服务器将把文字值排在匹配模式的前面，比

较具体的模式排在比较宽松的模式的前面。这意味着，当你从一台名为 boa.snake.net 的主机尝试连接某个 MySQL 服务器的时候，如果它的 user 数据表里有两个 Host 值分别是 boa.snake.net 和 %.snake.net 的数据行，它将优先使用前者而不是后者。同样的道理，%.snake.net 优先于%.net，%.net 又优先于%。IP 地址的匹配顺序也是如此。当某个客户从一台 IP 地址为 192.168.3.14 的主机尝试连接时，user 数据表里有着下列 Host 值——它们已按优先级从高到低进行了排序——的数据行都能与之匹配：

```
192.168.3.14
192.168.3.%
192.168.%
192.%
%
```

还有一个大家应该记住的原则：当服务器在 user 数据表里寻找匹配的数据行时，它先寻找 Host 值相匹配的数据行，找到后再检查该数据行的 User 值是不是也匹配。除此之外没有其他顺序。

13.2.3　一个关于权限的难题

在对来自用户的连接请求进行身份验证时，服务器将按特定的顺序去搜索 user 数据表里的数据行。看过本节讲述的故事之后，你就会理解我为什么说了解这个顺序很有用了。你还可以通过这个故事学会如何解决一个在新安装的 MySQL 系统里相当常见（因为它在各种 MySQL 邮件列表里出现的频率很高）的问题。故事从一位 MySQL 系统管理员刚安装好 MySQL 软件开始，此时的 user 数据表里通常只有对应于默认的 root 用户和匿名用户的数据行。任何一位稍有经验的系统管理员都会为 root 账户设置口令，但让匿名账户保持原样，因而没有任何口令的情况还是很常见的（除非他知道后面将要发生的事）。现在，假设那位系统管理员想创建一个新账户，好让某位用户能够从几台不同的主机连接 MySQL 服务器。最省事的办法是在创建新账户时在 GRANT 语句里使用“%”作为账户名的主机部分，这种设置可以让那位用户从任何地方连接 MySQL 服务器：

```
GRANT ALL ON sampdb.* TO 'fred'@'%' IDENTIFIED BY 'cocoa';
```

上面这条语句的本意是把 sampdb 数据库上的所有权限授予用户 fred，并让他能够从任何一台主机连接 MySQL 服务器，可是结果却很可能是这样的：用户 fred 能够从任何一台其他的主机连接 MySQL 服务器，唯独在服务器主机上不行！假设服务器主机的名字是 cobra.snake.net。当 fed 从名为 boa.snake.net 的主机远程连接 MySQL 服务器时，他的尝试就像预期的那样成功了：

```
% mysql -p -u fred -h cobra.snake.net sampdb
Enter password: cocoa
mysql>
```

可是，当 fred 试图从服务器主机 cobra.snake.net 进行本地连接时，却怎么连接不上，哪怕给出了正确的口令也不行：

```
% mysql -p -u fred -h localhost sampdb
Enter password: cocoa
ERROR 1045 (28000): Access denied for user 'fred'@'localhost'
(using password: YES)
```

只要 user 数据表有某个数据行为匿名用户默认创建的用户名字段为空，就会发生这样的问题。

这种数据行是由 Unix 平台上的 mysql_install_db 初始化脚本创建的，在适用于 Windows 平台的 MySQL 发行版本所收录的预初始化 user 数据表里也经常会看到它们。（12.1 节对初始状态的 user 数据表行做了细致的描述。）第二次连接尝试失败的原因是：当 MySQL 服务器对用户 fred 进行身份验证时，user 数据表里的一个匿名用户的数据行优先于用户 fred 的数据行得到了匹配。那个半路杀出的匿名用户数据行要求用户在连接时不使用口令，使用了口令（如本例中的 cocoa）的结果反倒是无法匹配。

为什么会发生这样的事情呢？若想找出问题的症结，需要知道 MySQL 权限数据表有着怎样的初始设置状态，以及服务器在验证客户连接时是如何使用 user 数据表行的。让我们回到我们的故事中来。在 Unix 环境下，当你在名为 cobra.snake.net 的主机上运行 mysql_install_db 脚本对权限数据表进行了初始化之后，user 数据表将包含由如下所示的 Host 值和 User 值构成的数据行：

```
+-----------------+------+
| Host            | User |
+-----------------+------+
| localhost       | root |
| cobra.snake.net | root |
| 127.0.0.1       | root |
| localhost       |      |
| cobra.snake.net |      |
+-----------------+------+
```

前 3 个数据行可以让人们作为 root 用户从本地服务器连接，后两个数据行可以让人们作为匿名用户从本地服务器连接。系统管理员使用前面给出的 GRANT 语句为用户 fred 创建了账户之后，user 数据表将包含如下所示的数据行：

```
+-----------------+------+
| Host            | User |
+-----------------+------+
| localhost       | root |
| cobra.snake.net | root |
| 127.0.0.1       | root |
| localhost       |      |
| cobra.snake.net |      |
| %               | fred |
+-----------------+------+
```

可是，在这里看到的数据行的排列顺序并不是服务器在验证连接请求时使用的顺序。正如此前讲过的，MySQL 服务器会以 Host 值为主、以 user 值为辅对 user 数据表里的数据行进行排序，把较为具体的值放在较为宽松的值的前面：

```
+-----------------+------+
| Host            | User |
+-----------------+------+
| localhost       | root |
| localhost       |      |
| 127.0.0.1       | root |
| cobra.snake.net | root |
| cobra.snake.net |      |
| %               | fred |
+-----------------+------+
```

Host 值是 localhost 的两个数据行排在了一起，User 值是 root 的那个数据行排在前面，这是因为较为具体的用户名优先于空白值。Host 值是 cobra.snake.net 的数据行也因为同样的道理而排在了一起。此外，因为这几个数据行的 Host 值都是具体的值，不带任何通配符，所以它们都排列在了 User 值是 fred 的那个数据行前面，后者的 Host 值里明显有一个通配符。于是，按照上述排序顺序，对应于匿名用户的那两个数据行都将优先于 fred 的数据行。

这样一来，当 fred 尝试从本地主机建立连接的时候，就会有一个有着空白用户名的数据行先于 Host 数据列包含着“%”值的那个数据行得到匹配。因为对应于匿名用户的数据行里的空白口令和 fred 的口令 cocoa 无法匹配，所以这次连接尝试只能以失败告终。这个故事还隐含着另外一种结局：fred 还是有可能从本地主机连接成功的，但连接成功的前提是他不能给出任何口令。如此建立的连接将使他被验证为一位匿名用户，因而无法获得与 fred@%账户相关联的权限。

这个故事告诉我们，虽然使用通配符来为一位需要从多台主机连接的用户创建账户非常方便，但因为 user 数据表里有对应于匿名用户的数据行，那位用户从本地主机连接时可能反而会遇到麻烦。

怎样才能解决这个问题呢？有两个办法。第一个办法是为 fred 再创建一个账户，把该账户的主机值明确地设置为 localhost：

```
GRANT ALL ON sampdb.* TO 'fred'@'localhost' IDENTIFIED BY 'cocoa';
```

这么做了以后，服务器对 user 数据表的排序结果将是如下所示的样子：

```
+-----------------+------+
| Host            | User |
+-----------------+------+
| localhost       | fred |
| localhost       | root |
| localhost       |      |
| 127.0.0.1       | root |
| cobra.snake.net | root |
| cobra.snake.net |      |
| %               | fred |
+-----------------+------+
```

现在，当 fred 从本地主机连接时，同时包含 localhost 和 fred 的那个数据行将先于匿名用户的数据行得到匹配。当他从任何其他主机连接时，同时包含%和 fred 的数据行将得到匹配。为 fred 创建两个账户的弊病是：以后只要你想改变他的权限或口令，将不得不修改两次。

第二个办法更容易：使用如下所示的 DROP USER 语句从 user 数据表删除匿名账户：

```
DROP USER ''@'localhost';
DROP USER ''@'cobra.snake.net';
```

user 数据表里剩下来的数据行将按以下顺序排列：

```
+-----------------+------+
| Host            | User |
+-----------------+------+
| localhost       | root |
| 127.0.0.1       | root |
| cobra.snake.net | root |
| %               | fred |
+-----------------+------+
```

现在，当 fred 尝试从本地主机连接时就会成功了，因为此时的 user 数据表里已经没有任何会先于他得到匹配的数据行了。

总体来说，我的建议是：如果想让自己的 MySQL 系统管理员生涯少些意外，就应该从初始权限数据表里把所有的匿名用户账户删除干净。在我看来，这些账户并没有多大用处，它们带来的问题往往比它们带来的方便更多。

本节讲述的小故事只描述了一种特殊情况，但其中蕴含的道理却是通用的。如果某给定账户的权限没有你预期的那样发挥作用，就应该去查看一下权限数据表，看里面是否存在 Host 值比那位用户的更加具体的数据行，那些数据行会在该用户尝试连接 MySQL 服务器时抢先得到匹配。你可能需要把那位用户的数据行设置得更加具体，或是增加一个数据行来专门应对。

13.2.4 应该回避的权限数据表风险

本节将介绍一些在授予权限时应该注意回避的问题，以及几种不明智的设置可能带来的风险。

避免创建匿名账户。即使匿名账户本身没有足够的权限去造成直接的损害，但允许某个用户匿名登录无疑会让该用户有机会在你的数据库系统里四处游荡，并收集诸如你都拥有哪些数据库和数据表之类的信息，或是让他有机会使用 SHOW STATUS 和 SHOW VARIABLES 等语句去监视你的服务器的工作情况。

找出没有口令的账户并删除或是给它们加上口令。下面这个查询可以找出这样的账户：

```
mysql> SELECT Host, User FROM mysql.user WHERE Password = '';
```

找出口令散列的格式早于 MySQL 4.1 版的账户，并把它们改为从 MySQL 4.1 版开始启用的更安全的口令散列格式。老格式口令值的长度是 16 个字符且不以“*”字符开头，所以你可以使用下列语句之一来找出它们：

```
mysql> SELECT Host, User FROM mysql.user WHERE LENGTH(Password) = 16;
mysql> SELECT Host, User FROM mysql.user WHERE Password NOT LIKE '*%';
```

请注意，实施这一安全措施是有前提的，那就是所有用来连接服务器的客户程序都应该来自 MySQL 4.1 或更高版本，因而能够使用新的口令机制去验证用户的身份。在这个前提下，只要确保服务器在启动时没有使用--old-password 选项，再使用 SET PASSWORD 语句给所有使用老口令格式的账户重新设置口令，就可以让所有的账户都使用新的口令格式了。为了进一步提高安全性，还可以使用--secure-auth 选项来启动服务器。如果不这么做，客户就可以先利用 OLD_PASSWORD()函数把它的口令重新设置回老格式，再使用老口令去连接服务器了。--secure-auth 选项的作用是确保只有使用了新格式口令的客户才有可能连接成功。

在设置账户的时候，如果没有必要，就应该尽量避免在主机名字段里使用匹配模式。若允许某给定用户从更多的主机进行连接，会让恶意人员有机会从更多的地点假冒该用户入侵你的系统。

在授予超级用户权限的时候一定要谨慎再谨慎。不要在 user 数据表里进行授权，因为如此授予的权限都将是全局级的，获得授权的用户将能影响服务器的操作或是有权访问任何数据库。正确的做法是只在特定级别进行授权，这样才能确保用户只能访问特定的数据库或是诸如数据表和存储例程之类的数据库对象。

千万不要把容纳着权限数据表的 mysql 数据库的访问权限授予无关人员。有权访问该数据库的用户可以通过修改其中的数据表而获得访问任何其他数据库的权限。事实上，授权某个用户修改 mysql

数据库无异于把所有的全局级权限统统授予给了该用户。如果该用户可以直接修改数据表，那就可以执行任何你能想象得到的账户管理语句。

慎重对待 GRANT OPTION 权限。两个有着不同的权限、但都有 GRANT OPTION 权限的用户可以让对方的访问权限变得更强大。

FILE 权限特别危险，千万不要轻易地授予他人。下面的例子是某个拥有 FILE 权限的用户可以做到的事情：

```
CREATE TABLE etc_passwd (pwd_entry TEXT);
LOAD DATA INFILE '/etc/passwd' INTO TABLE etc_passwd;
```

执行完这两条语句，该用户只要执行下面这条 SELECT 语句就可以看到服务器主机的口令文件的内容了：

```
SELECT * FROM etc_passwd;
```

上面那条 LOAD DATA 语句里的/etc/etc_passwd 还可以替换为服务器主机上的任何一个全局可读文件的名字。如果某个用户是从一台远程主机连接的，若把 FILE 权限授予他，他就能通过网络访问到服务器主机的文件系统的很大一部分。

在一个 MySQL 数据目录的访问权限被设置得不够安全的系统上，FILE 权限往往会成为恶意人员盗取数据库内容的帮凶。这也正是应该只让服务器读取数据目录的内容的理由之一。如果对应于某个数据库或数据表的文件是全局可读的，那么不仅任何一位在服务器主机上有登录账户的人可以读取它们，任何一位拥有 FILE 权限的 MySQL 客户用户也可以读取它们。请看下面给出的演示过程。

(1) 创建一个数据表并让它包含一个 LONGBLOB 数据列：

```
USE test;
CREATE TABLE tmp (b LONGBLOB);
```

(2) 把与你打算盗取的数据表相对应的各个文件的内容依次读取到这个 tmp 数据表里。比如说，我们不妨假设某个用户在一个名为 other_db 的数据库里有一个名为 x 的 MyISAM 数据表，该数据表对应着 3 个文件，即 x.frm、x.MYD 和 x.MYI。你可以像下面这样依次读取那些文件的内容并把它们复制到 test 数据库中的相关文件里去：

```
LOAD DATA INFILE './other_db/x.frm' INTO TABLE tmp
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
SELECT * FROM tmp INTO OUTFILE 'x.frm'
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
DELETE FROM tmp;
LOAD DATA INFILE './other_db/x.MYD' INTO TABLE tmp
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
SELECT * FROM tmp INTO OUTFILE 'x.MYD'
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
DELETE FROM tmp;
LOAD DATA INFILE './other_db/x.MYI' INTO TABLE tmp
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
SELECT * FROM tmp INTO OUTFILE 'x.MYI'
  FIELDS ESCAPED BY '' LINES TERMINATED BY '';
```

(3) 在执行完这些语句之后，test 数据库所在的子目录将包含名字是 x.frm、x.MYD 和 x.MYI 的文件。换句话说，test 数据库将多出一个名为 x 的数据表，其内容与 other_db 数据库里的那个同名数据表一模一样。

要想避免有人以同样的方式去攻击系统上的用户们的数据表，请务必按照 13.1.2 节的步骤来设置数据目录及其内容的访问权限。此外，尽量避免把 SHOW DATABASE 权限授予无关人员，并使用 --skip-show-database 选项来启动服务器。这可以防止个别用户对他们无权访问的数据库使用 SHOW DATABASES 和 SHOW TABLES 语句，不让他们知道他们无权访问的数据库和数据表都有哪些。

如果使用操作系统的 root 账户来运行服务器，FILE 权限的危险性将进一步扩大。那么做无论如何都是不明智的，再与 FILE 权限结合起来就更不明智了。因为操作系统的 root 账户可以在文件系统中的任何地方创建文件，所以拥有 FILE 权限的 MySQL 用户也具备那样的能力，哪怕他是一位从某个远程主机连接的用户。虽说 MySQL 服务器会拒绝创建一个已经存在的文件，但通过 MySQL 服务器创建的新文件有时也可以改变服务器主机的操作行为或是破坏其安全性。比如说，如果在你的系统上/etc/resolv.conf、/etc/hosts.equiv、/etc/host.lpd 或者 ect/sudoers 文件中有一个不存在，一个有权通过 MySQL 服务器创建它们的用户就可以极大地改变服务器主机的行为。如果想避免这些问题，就千万不要使用 root 账户去运行服务器。（请参阅 12.2.1 节的第 1 小节。）

PROCESS 和 SUPER 权限应该只授予给值得充分信任的 MySQL 账户。一位具备 PROCESS 权限的用户可以使用 SHOW PROCESSLIST 语句查看到 MySQL 服务器正在执行的语句的文本。这意味着那位用户可以通过嗅探其他用户的操作而窥视到他人的私密信息。具备 SUPER 权限的用户可以“杀死”其他用户正在执行的语句，干扰他们的操作活动。SUPER 权限还可以让用户有权整理日志文件或是采取其他会破坏服务器操作的行动。

不要把 RELOAD 权限授予不需要它的人。RELOAD 权限可以让用户有权执行 FLUSH 和 RESET 语句，而这两条语句很可能会被恶意滥用。

- 二进制日志文件和中继日志文件是使用一组按顺序编号的名字创建的。如果已经配置你的服务器启用了二进制或中继日志功能，那么每执行一次FLUSH LOGS语句就会创建出该序列里的下一个日志文件。如果某个具备RELOAD权限的的用户频繁进行日志刷新操作，就可以让服务器创建出大量的日志文件。
- 具备RELOAD权限的用户可以通过执行FLUSH PRIVILEGES或FLUSH USER_RESOURCES语句加载权限数据表的办法打破服务器的资源管理机制。这两条语句都可以把各资源管理计数器重置为零。
- 频繁执行FLUSH TABLES语句会导致服务器清除它的数据表缓存，这条语句会让MySQL因为无法正常使用这个缓存区而性能下降。同样，RESET QUERY CACHE也会破坏查询缓存的作用。
- RESET MASTER LOGS语句会导致复制主服务器强行删除它所有的二进制日志文件，不管它们是否仍在使用。恶意删除那些信息将破坏复制机制的完整性。

ALTER 权限也可能被坏人用来干坏事。假设你的本意是想让某个用户去访问 table1 而不是 table2。但另外一个具备 ALTER 权限的用户可以通过使用 ALTER TABLE 语句把 table2 重命名为 table1 的办法歪曲你的本意。

13.3 加密连接的建立

MySQL 支持使用基于 SSL 的加密连接。在默认的情况下，启用了 SSL 支持功能的 MySQL 安装将允许客户选择是否使用加密连接。MySQL 数据库系统的管理员还可以用 GRANT 语句规定某些账户

必须使用加密连接。一般来说，普通的非加密连接有着较高的性能。加密连接虽然安全但速度较慢，这是因为对数据进行加密会加重系统的计算负担。

需要提醒大家注意的是，对于通过 Unix 套接字文件、命名管道、共享内存或 IP 地址 127.0.0.1（本地主机的网络回馈接口）建立的对本地主机的连接，使用 SSL 没有什么意义。在这些连接上传输的信息根本不可能离开本地主机。SSL 的真正价值体现在所传输的信息会经过某个你怀疑正被人嗅探的网络时。

下面是在服务器与客户程序之间建立 SSL 加密连接的基本步骤。

(1) 服务器和客户程序都已经编译有 SSL 支持机制。

(2) 在启动 MySQL 服务器时用有关选项告诉它到哪儿能找到它的证书文件和密钥文件。这些文件是建立加密连接所不可缺少的。

(3) 要想让某个客户程序建立加密连接，必须在调用这个客户程序时用有关选项告诉它到哪儿能找到你自己的证书文件和密钥文件。

接下来将对以上过程做进一步的讨论。

首先，MySQL 发行版本必须已经把 SSL 支持机制编译在其中了。你可以设法弄一份编译有 SSL 支持机制的二进制发行版本，也可以自行编译 MySQL 软件的源代码。如果是自行编译 MySQL 软件，千万不要忘记在编译配置阶段提供必要的选项（比如说，在 Unix 系统上，必须提供`--with-*ssl` 选项。详见 A.4.3 节的第 3 小节）。在启动了具备 SSL 支持机制的 MySQL 服务器之后，可以用 `mysql` 连接它并发出下面这个查询命令去检查它是否支持 SSL：

```
mysql> SHOW VARIABLES LIKE 'have_ssl';
+---------------+----------+
| Variable_name | Value    |
+---------------+----------+
| have_ssl      | DISABLED |
+---------------+----------+
```

如果看到的是 `DISABLED` 或 `YES`，就说明它支持 SSL。`DISABLED` 的意思是具备 SSL 支持但尚未启用它。这不是什么大问题。启用 SSL 支持功能所需要的文件将在下面讨论。

在正确启用了 MySQL 的 SSL 支持机制之后，MySQL 服务器和它的客户就可以安全通信了。为建立安全连接，服务器端和客户端都要准备好 3 个文件。

- 证书签发机构（Certificate Authority，简称CA）证书，指的是一个值得信赖的第三方，它的证书将被用来验证客户端和服务器端提供的证书。CA证书可以从某个商业机构购买，也可以自行生成。
- 证书文件，连接的一方向另一方证明自己身份的文件。
- 密钥文件，用来对在连接上传输的数据进行加密和解密。

服务器端的证书文件和密钥文件必须首先安装。如果你还没有自己的这些文件，可以在 `sampdb` 发行版本的 ssl 子目录里找到几个试用样本文件。

- `ca-cert.pem`——CA证书。
- `server-cert.pem`——MySQL服务器的证书。
- `server-key.pem`——MySQL服务器的公共密钥。

先把这几个文件复制到 MySQL 服务器的数据目录里，然后往它将在启动时读取的某个选项文件（比如 Unix 系统上的/etc/my.cnf 文件或 Windows 系统上的 C:\my.ini 文件）里的`[mysqld]`选项组里加上

几个选项，这些选项的作用是给出证书文件和密钥文件的路径名。比如说，如果数据目录是/usr/local/mysql/data，有关选项将如下所示：

```
[mysqld]
ssl-ca=/usr/local/mysql/data/ca-cert.pem
ssl-cert=/usr/local/mysql/data/server-cert.pem
ssl-key=/usr/local/mysql/data/server-key.pem
```

如果你愿意，也可以把证书文件和密钥文件放在其他地方，但必须确保那个子目录只能由服务器访问。修改完选项文件之后，重新启动服务器。

现在，服务器已经允许客户使用加密连接了，have_ssl 系统变量的值也应该变成 YES。不过，客户程序仍只能使用未加密的连接去连接服务器。要想让客户程序使用加密连接，就要把供客户程序使用的证书文件和密钥文件也准备好。sampdb 发行版本的 ssl 子目录也提供了几个这样的文件，你可以使用同一份 CA 证书文件（ca-cert.pem）。客户端的证书文件和密钥文件的文件名分别是 client-cert.pem 和 client-key.pem。把这 3 个文件复制到你个人账户下的某个子目录里，然后往客户程序将在它开始执行时读取的某个选项文件（比如 Unix 系统上的用户登录子目录中的.my.cnf 文件）里加上几个选项，以便让客户程序知道它们都放在哪里。

举个例子，如果想让 mysql 程序使用加密连接，就得先把必要的 SSL 文件复制到主目录/home/paul 里，再把以下各行添加到我的.my.cnf 文件里去：

```
[mysql]
ssl-ca=/home/paul/ca-cert.pem
ssl-cert=/home/paul/client-cert.pem
ssl-key=/home/paul/client-key.pem
```

你可以如我这样去设置你们自己的账户。请注意，你应该确保证书文件和密钥文件只能由你本人访问。在.my.cnf 文件里给出 SSL 文件的存放路径，执行 mysql 程序并发出一个\s 或 STATUS 命令，如果在输出里有一个 SSL 行，就说明连接是加密的：

```
mysql> status;
--------------

mysql  Ver 14.14 Distrib 5.1.25-rc, for pc-linux-gnu (i686)

Connection id:          5
Current database:
Current user:           sampadm@localhost
SSL:                    Cipher in use is DHE-RSA-AES256-SHA
...
```

还可以用如下所示的数据库查询命令去查看与 SSL 有关的服务器状态变量：

```
SHOW STATUS LIKE 'Ssl%';
```

[mysql]选项组里与 SSL 有关的那几个选项将使 mysql 程序默认使用 SSL 连接。如果把那几行改为注释，或者从选项文件里删掉了它们，mysql 程序将使用普通的非加密连接。此外，如果像下面这样调用 mysql 程序，它将忽略与 SSL 有关的选项：

```
% mysql --skip-ssl
```

如果想让其他客户程序也使用 SSL 加密连接，可以把[mysql]选项组里的 SSL 选项复制到对应于

其他客户程序的选项组里。不过，把 SSL 选项放到[client]选项组里不见得是个好主意，那会让不知道如何使用 SSL 的客户程序不能正常使用。如果非得把那些选项放到[client]选项组里，应该给它们加上 loose-前缀，这可以让那些不支持 SSL 的客户程序忽略它们。

如果不想在选项文件里列出 SSL 选项，也可以把它们写在命令行上。比如说，在我的 SSL 文件所在的子目录里，可以像下面这样去运行 mysql 程序（注意，这条命令必须在同一行输入）：

```
% mysql --ssl-ca=ca-cert.pem --ssl-cert=client-cert.pem
    --ssl-key=client-key.pem
```

不过，如果经常使用 SSL 加密连接，在命令行上给出 SSL 选项就太麻烦了。

sampdb 发行版本中的证书文件和密钥文件完全能够让你建立加密连接。不过，因为谁都能获得它们，所以这种加密连接在安全方面多少要打些折扣。在用这些文件检查完 SSL 是否能够正确工作之后，最好把它们替换成你自己生成的文件。制作证书文件和密钥文件的具体步骤请参见 sampdb 发行版本中的 ssl/README.txt 文本文件。当然了，也可以考虑购买一份商业证书。

以上讨论的前提是有关账户能够选择是否使用 SSL。MySQL 管理员还可以禁止某个账户使用未加密连接，即必须使用 SSL。对新建账户和现有账户均可这样设置。

先说新建账户。像往常一样发出一条 GRANT 语句，但要增加一个 REQUIRE 子句来规定该账户必须遵守某些限制。比如说，若想创建一个名为 laura 的用户，他将从主机 rat.snake.net 连接到主机 cobra.snake.net 上的服务器以访问 finance 数据库。如果只要求他建立的连接是加密的就行，请使用下面这条语句：

```
GRANT ALL ON finance.* TO 'laura'@'rat.snake.net'
  IDENTIFIED BY 'moneymoneymoney'
  REQUIRE SSL;
```

如果还想更安全点儿，可以使用 REQUIRE X509 子句。这样，用户 laura 在连接时就必须提供一份合法的 X509 客户证书（这个证书文件对应于 ssl-cert 选项）。只要这份证书是合法的，它的内容是什么无关紧要。如果想做出更多的限制，可以在 REQUIRE 子句里组合使用 CIPHER、ISSUER 或 SUBJECT 等限定词。CIPHER 规定了加密连接必须使用哪一种加密密码。ISSURE 和 SUBJECT 则分别规定了客户证书必须签发自哪一个 CA 机构，以及证书的持有者必须是谁。这些限定词对已经合法的证书做出了更细致的规定，只有满足所有规定的证书才能用来建立 SSL 加密连接。请看下面这条 GRANT 语句，它要求客户证书必须由指定 CA 机构签发，而且必须使用 EXP1024-RC4-SHA 加密算法：

```
GRANT ALL ON finance.* TO 'laura'@'rat.snake.net'
  IDENTIFIED BY 'moneymoneymoney'
  REQUIRE ISSUER '/C=US/ST=WI/L=Madison/O=sampdb/OU=CA/CN=sampdb'
  CIPHER 'EXP1024-RC4-SHA';
```

如果想修改某个现有账户，让它今后必须使用 SSL 连接，就要使用格式如下所示的 GRANT USAGE 语句，其中的 require_options 代表你想要设置的 SSL 属性：

```
GRANT USAGE ON *.* TO 'user_name'@'host_name' REQUIRE require_options;
```

GRANT USAGE ON *.*不会改变现有账户的现有权限，只修改与 SSL 有关的属性。

可以用 GRANT USAGE 语句的 REQUIRE NONE 子句来撤销对某个现有账户必须使用 SSL 连接的规定：

```
GRANT USAGE ON *.* TO 'user_name'@'host_name' REQUIRE NONE;
```

如果正在使用 PHP、Perl 等语言的 MySQL API 编写应用程序，应用程序能否支持 SSL 将取决于两个因素：API 和 API 的底层客户端库。只有客户端库具备 SSL 支持，用它编写出来的应用程序才可以使用 SSL 连接服务器。在此基础上，API 软件包的版本必须足够新才能使用客户端库的 SSL 能力。比如说，PHP 语言的 `mysql` 扩展包支持 SSL 连接，但较早的 `mysql` 扩展包不支持。

第 14 章

MySQL 数据库的维护、备份和复制

14

如果 MySQL 从你第一次安装好以后就一直稳定地运行，那当然是最理想的了。但问题会因为种种原因而发生，从意外停电到硬件故障到 MySQL 服务器非正常关停（比如你用 `kill -9` 终止 MySQL 服务器或服务器主机崩溃等）。这些事情有很多是你无法控制的，它们往往会对数据库里的数据表造成破坏，通常是在数据表的写操作期间发生意外而导致的。

在这一章里，我将向大家介绍一些回避风险和万一发生灾难后的应对措施，涉及数据库备份技术、数据表检查和修复技术，以及数据丢失后的恢复技术等。本章还将介绍一些用来把数据库转移到另一个服务器的数据库复制技术。这种事情经常会发生，并且与数据库备份技术有很多相似之处。这里还将讨论另一种复制技术（replication），即建立一个从服务器（slave server）来实时地复制主服务器（master server）上的数据，当主服务器上的数据发生变化时，同样的变化将同步地发生在从服务器上。从效果上看，从服务器就像是主服务器在镜子里的影像，两者随时保持着一致。这种机制还可以用于其他用途，比如说，数据库管理员可以把客户对主服务器的一部分访问负载分流到从服务器以减轻主服务器上的负担；再比如说，主服务器往往需要保持 24 小时连续运转，为进行数据库备份工作而关停主服务器恐怕不现实，而为此暂停或是关停从服务器在决策和操作两方面都要容易得多。

14.1　数据库预防性维护工作的基本原则

本节汇总了一些与数据库系统的预防性维护工作有关的基本原则。随后的几个小节将详细讨论如何根据这些原则去完成各项具体工作。

为预防可能发生的数据库故障，应该采取以下措施。

- 激活MySQL服务器的自动恢复能力。
- 有计划地安排一些预防性的维护工作，定期对数据表进行检查。数据表的日常检查工作能帮助你及时发现和纠正那些小问题而不是让它们变得更糟糕。
- 制定一份数据库备份计划。一旦发生最坏情况并需要你去面对重大的系统灾难时，你将需要备份来进行恢复工作。还应该启用二进制日志，它们记载着你在备份之后又对数据库的内容进行过哪些修改（请参阅12.5.4节）。二进制日志对于备份和复制（replication）有显著帮助，因启用二进制日志而导致的额外开销非常之小（约1%），所以没有理由不激活之。

万一遇到数据表损坏或数据丢失问题，请按以下原则处理。

- 检查数据表，尽可能对发现的问题进行修复。MySQL的数据表修复能力能应付很多小破坏。
- 如果对数据表进行的检查和修复仍不能使你的MySQL服务器恢复运行，就要用你的备份和二进制日志来进行数据恢复工作了。先用备份把数据表恢复到当初进行备份时的状态，再根据二进制日志重新应用备份前进行的修改，把数据表恢复到灾难发生之前的状态。

用来完成以上任务的工具包括 MySQL 服务器本身的能力和 MySQL 发行版本自带的几种工具程序。

- 在MySQL服务器启动时，事务型存储引擎将自动进行数据表检查和恢复处理。在此基础上，MySQL管理员还可以激活MyISAM存储引擎的数据表自动修复功能。当你在服务器崩溃之后重新启动MySQL服务器时，这种能力将非常有用。
- 使用mysqldump或mysqlhotcopy程序为数据库制作备份，这些备份是日后恢复数据库所必需的。
- 可以用CHECK TABLE和REPAIR TABLE等SQL语句让MySQL服务器根据需要执行几种数据表维护操作。mysqlcheck工具程序为这些SQL语句提供了一个命令行操作界面。myisamchk工具程序也能对数据表进行检查并对它们进行多种修复。

有些程序（比如 mysqlcheck 和 mysqldump）需要有 MySQL 服务器的配合才能使用。它们必须像其他客户（程序）那样先连接到 MySQL 服务器，然后再发出 SQL 语句去告诉服务器应该执行什么样的数据表维护操作。与此相反的是 myisamchk 程序，它是一个可以直接操作数据表文件的独立程序。不过，因为 MySQL 服务器在运行时也要访问那些文件，myisamchk 程序在完成有关操作时难免会与 MySQL 服务器发生竞争，所以必须采取措施防止它们相互干扰。比如说，在使用 myisamchk 程序修复某个数据表时，一定要保证 MySQL 服务器不会在此期间也去访问它，否则，会导致比你正试图修复的问题更加严重的问题！

本章所涉及的几项管理任务——从制作备份到进行数据表修复——都需要与 MySQL 服务器合作，所以我们将从如何与 MySQL 服务器进行协调开始讲起，然后讨论如何未雨绸缪、如何制作备份和如何在必要时使用各种数据库恢复技术。

在 Unix 系统上，当需要直接操作 MySQL 数据目录中的数据表文件或其他文件时，你应该以 MySQL 管理员登录后再进行这些操作，这将保证你有足够的权限去使用这些文件。在这本书里，我们给这个账户起的名字是“mysql”。当然了，以 root 用户身份也可以访问这些文件，但在这种情况下，一定要保证有关文件在你开始操作之前和完成操作之后的访问模式和属主是相同的。

本章所涉及的各 SQL 语句和工具程序的完整选项清单见附录 E 和附录 F。

14.2　在 MySQL 服务器运行时维护数据库

执行数据库维护操作的办法之一是连接 MySQL 服务器并告诉它应该做什么事情。比如说，如果需要对某个 MyISAM 数据表进行数据完整性检查或进行修复，一种做法是发出 CHECK TABLE 或 REPAIR TABLE 语句（或运行 mysqlcheck 程序）让 MySQL 服务器去做这项工作，也就是让 MySQL 服务器去访问代表着这个数据表的.frm、.MYD 和.MYI 文件。一般说来，这种做法是最好的：在执行维护操作时，可能发生的数据表访问冲突都将由 MySQL 服务器负责解决，用不着你去操心。

执行数据库维护操作的另一种办法，是使用一个不依赖于 MySQL 服务器的外部程序，但这个办法需要由你来协调各有关数据表上的访问冲突。比如说，检查或修复 MyISAM 数据表的工作还可以通过运行 myisamchk 程序的办法来进行，该程序将直接打开有关的数据表文件而不是通过 MySQL 服务

器去做这些事情。在 myisamchk 程序正在访问数据表文件时，你必须阻止 MySQL 服务器去修改那个数据表。如果不这样做，数据表就可能因为 myisamchk 程序和 MySQL 服务器发生竞争而遭到破坏甚至无法继续使用。很明显，让 MySQL 服务器和 myisamchk 程序同时对同一个数据表进行写操作是不好的，但即使它们是一个在读而另一个在写也是不好的——正在读取数据表文件的程序会因为另一个程序对数据表的修改而被弄糊涂。

在其他一些上下文里也需要阻止 MySQL 服务器去访问数据表。

- 在使用myisampack程序压缩某个MyISAM数据表的时候。
- 在重新安置某个MyISAM数据表的数据文件或索引文件的时候。
- 在重新安置一个数据库的时候。
- 有些备份技术需要制作数据表文件的副本，而要想保证备份文件的一致性，就必须防止MySQL服务器在备份过程中修改数据表。
- 有些数据恢复技术需要把被损坏的数据表替换为完好的备份。在对某给定数据表进行这种替换的过程中，绝不能允许MySQL服务器去访问那个数据表。

如果不想让 MySQL 服务器打扰你，最有效的办法就是把它彻底关停。如果 MySQL 服务器根本没有运行，它当然不会去访问你正在处理的数据表。但 MySQL 管理员通常都不太愿意把自己负责的 MySQL 服务器彻底关停，因为这将使所有的数据库和数据表——而不仅仅是你打算检查或修复的那几个——都无法使用。

如果不想关停 MySQL 服务器，又不想让运行中的服务器和你正在使用的外部程序相互干扰，就必须使用某种锁定机制来与服务器进行协调。MySQL 服务器提供了两种锁定机制。

- 内部锁定机制。MySQL服务器使用这一机制来防止来自不同客户程序的查询请求相互混杂和干扰，比如防止来自某个客户程序的SELECT查询被来自另一个客户程序的UPDATE语句打断。这种内部锁定机制也可以用来阻止其他客户访问我们正在使用一个外部程序对之进行处理的数据表。
- 外部锁定机制。MySQL服务器使用这一机制来防止其他程序修改它正在使用的数据表文件。这种外部锁定机制是以操作系统在文件系统级的锁定能力为基础的。人们给MySQL服务器加上外部锁定机制是为了让它能够与诸如myisamchk之类的工具程序在数据表检查操作期间进行协调。但可惜的是，MySQL服务器的外部锁定机制在某些系统上工作得不太可靠。另一个限制是MySQL服务器的外部锁定机制只能用来协调对数据表文件进行只读访问的操作，如数据表检查操作；不能用来协调对数据表文件进行读/写访问的操作，如数据表修复操作。MySQL服务器的外部锁定机制是以操作系统的文件锁定机制为基础的，但因为myisamchk程序在修复数据表的时候会先把数据表文件复制为一些新文件，等修复工作完成后再用新文件去替换原来的数据表文件，MySQL服务器对那些新文件将一无所知，所以即便你想利用文件锁定机制去协调在此期间可能发生的访问冲突，你的努力也将徒劳无功。

接下来的讨论只涉及如何使用内部锁定机制与 MySQL 服务器协调有关数据表文件上的访问操作。至于外部锁定机制，因为它本身存在的不足，我就不对它做进一步讨论了。

14.2.1　以只读方式或读/写方式锁定一个或多个数据表

说到如何利用 MySQL 服务器的内部锁定机制去阻止它访问你正在处理的数据表，其基本思路是用 mysql 程序连接到 MySQL 服务器并发出一条 LOCK TABLE 语句锁定你打算使用的数据表，然后把

这个 mysql 会话闲置在那里（这个 mysql 会话只用来锁定那个数据表），用数据表文件做你想做的事情。在完成那些工作之后，切换回 mysql 会话并释放刚才锁定的数据表，让 MySQL 服务器可以重新使用它。

应该以哪种方式去锁定数据表取决于你将对数据表文件进行读操作还是进行读/写操作。

- 如果只是检查或者复制某些文件，以只读方式锁定它们就足够了。
- 如果需要修改某些文件，比如需要对数据表进行修复或者需要用完好的文件替换被损坏的文件时，应该以读/写方式锁定它们。

这两种锁定方式使用 LOCK TABLE 和 UNLOCK TABLE 语句来锁定数据表和解除锁定。它们还使用 FLUSH TABLE 语句通知 MySQL 服务器把挂起的改动写入硬盘，这条语句的另一个作用是告诉 MySQL 服务器在下次访问那个数据表的时候需要重新打开它。FLUSH TABLE 语句需要我们提供一个数据表名字作为参数，它将只把给定数据表挂起的改动信息写入硬盘文件。

上述基本思路的关键是必须在同一个 mysql 会话任务里执行所有的 LOCK、FLUSH 和 UNLOCK 语句。你不能使用 mysql 程序连接到 MySQL 服务器并锁定某个数据表然后立刻退出，试图稍后再次连接 MySQL 服务器去解除刚才的锁定了。这是行不通的，因为当你退出 mysql 程序时，服务器将自动解除你设置的锁定并认为它又能随意使用数据表了，而这意味着你对数据表文件的操作将不再安全。

执行锁定操作的一个简单方法是同时打开两个窗口——其中之一用来运行 mysql 客户程序，另一个用来对数据表文件进行处理。如果没有使用一个窗口化的操作环境，在需要直接对数据表文件进行操作的时候可以利用 shell 的作业调度命令去挂起和恢复 mysql 程序的运行。

这里描述的利用 MySQL 服务器的内部锁定机制去锁定给定数据表的办法仅适用于那些把每个数据表分别表示为一组相关文件的存储引擎（比如 MyISAM 存储引擎），不适用于把多个数据表的信息保存在同一个给定文件里的存储引擎（比如 InnoDB 或 Falcon 存储引擎）。比如说，在默认的情况下，InnoDB 存储引擎会把所有的 InnoD 数据表都保存在构成其共享表空间的文件里。（即使把 InnoDB 存储引擎配置成给每个数据表单独创建一个表空间的样子，它也会把每个数据表的一些信息保存在它的数据字典里，而数据字典总是保存在共享表空间里的。）

1. 以只读方式锁定一个数据表

只读锁定方式适用于只需对数据表文件进行读操作的场合，比如制作数据表文件的副本或者检查其数据完整性。对于这类操作，以只读方式锁定就已经足够了；MySQL 服务器将替你阻止其他客户程序去修改它，但仍允许其他客户程序从中读取数据。在需要对数据表做出修改的场合，不要使用这种锁定方式。

(1) 在窗口 A 里执行 mysql 程序，然后用以下语句申请一个读操作锁并把数据表在内存里的信息写入磁盘：

```
% mysql db_name
mysql> LOCK TABLE tbl_name READ;
mysql> FLUSH TABLE tbl_name;
```

用 LOCK TABLE 语句申请的读操作锁将阻止其他客户在你检查给定数据表的同时往它里面写入数据和修改它。FLUSH 语句将使 MySQL 服务器关闭有关的数据表文件，而文件关闭操作将把缓存在内存里的未写信息写入磁盘。

(2) 把 mysql 程序闲置在那儿，切换到窗口 B 操作数据表文件。比如说，你可以用下面这条命令去检查一个 MyISAM 数据表：

```
% myisamchk tbl_name
```

请注意，只有当你的当前目录是给定数据表的数据库目录时才能使用如上所示的命令。如果不是这样，就必须在数据表的名字前面加上它的路径名，如下所示：

```
% myisamchk /usr/loacl/mysql/data/tbl_name
```

这个例子只是为了演示有关概念。你实际使用的命令将取决于你们想要完成的数据库维护操作。

(3) 完成对数据表的处理之后，切换回窗口 A 里的 mysql 会话任务并解除对数据表的锁定：

```
mysql> UNLOCK TABLE;
```

在对数据表进行处理的过程中，你们很可能会发现还需要对它做进一步的处理。比如说，在用 myisamchk 程序检查某个数据表的过程中，很可能会发现一些需要纠正的问题。如果纠正工作需要以读/写方式去锁定那个数据表，就需要用下一小节介绍的步骤。

2. 以读/写方式锁定一个数据表

读/写锁定方式适用于需要对数据表文件的内容进行修改的场合，比如修复某个数据表。对于这类操作，必须申请一个写操作锁才能完全阻止 MySQL 服务器在你对数据表进行处理的时候去访问它。

为修复数据表而进行的锁定与为检查数据表而进行的锁定有两个不同之处。第一，必须以写方式进行锁定而不是以只读方式进行锁定。你将对数据表进行修改，所以绝不能让 MySQL 服务器去访问它。第二，在处理完数据表之后必须再发出一条 FLUSH TABLE 语句。有些操作——比如用 myisamchk 程序去修复数据表——会创建出一个新的索引文件，如果你没有刷新数据表缓存，MySQL 服务器就不会知道它的存在。以读/写方式锁定某个数据表的具体步骤如下所示。

(1) 在窗口 A 里执行 mysql 程序，然后用以下语句申请一个写操作锁并刷新数据表：

```
% mysql db_name
mysql> LOCK TABLE tbl_name WRITE;
mysql> FLUSH TABLE tbl_name;
```

(2) 把 mysql 程序闲置在那儿，切换到窗口 B 去操作数据表文件。比如说，你可以用下面这条命令去修复一个 MyISAM 数据表：

```
% myisamchk --recover tbl_name
```

这个例子只是为了演示有关概念。实际使用的命令将取决于你想要完成的数据库维护操作。（为预防万一，最好先制作一份数据表文件的副本。）

(3) 完成对数据表的处理之后，切换回窗口 A 里的 mysql 会话任务，再次刷新数据表，然后解除对数据表的锁定：

```
mysql> FLUSH TABLE tbl_name;
mysql> UNLOCK TABLE;
```

14.2.2　以只读方式锁定所有的数据库

要想阻止客户程序修改任何一个数据表，最简单的办法莫过于以只读方式锁定所有数据库里的所有数据表。可以用下面这些语句来做到这一点：

```
mysql> FLUSH TABLES WITH READ LOCK;
mysql> SET GLOBAL read_only = ON;
```

FLUSH 语句将申请一个全局性的读操作锁，SET 语句将在其他客户释放它们锁定的所有数据表并

完成所有正在执行的事务之后才开始执行，在此之前将一直处于被阻塞状态。在 SET 语句执行完毕之后，其他客户将只能进行读操作而不能对数据表做任何修改。

下面这些语句将解除刚才设置的锁定：

```
mysql> SET GLOBAL read_only = OFF;
mysql> UNLOCK TABLES;
```

当数据表以这种方式被锁定时，其他客户只能从它们读取数据，但不能进行任何修改。当你制作整个 MySQL 数据目录的副本时，用这个办法让 MySQL 服务器保持“沉默”是最合适的了。但从另一方面讲，这对想要修改数据表的其他客户来说可不太友好，所以你一定要在完成任务后立刻解除这种锁定。请注意，这个办法对某些操作来说还不够，比如为某个事务型存储引擎所管理的所有数据表制作一份二进制备份，因为该存储引擎可能有一些尚未全部完成的事务只是部分地写入它的日志文件里。那样的操作需要你彻底关停 MySQL 服务器才能保证所有的东西都已经被转储干净和所有的文件都已被关闭。

14.3 预防性维护

MySQL 管理员的职责之一是保证数据库完好无损。本小节将向大家介绍一些这方面的基本策略。

- 充分利用MySQL服务器本身的自动恢复能力。
- 有计划地安排一些预防性的维护工作，定期对数据表进行检查。
- 定期对数据库进行备份。这样，万一数据库遭到了无可挽回的破坏或丢失，也有最后的救命绝招。

前两项将在这里讨论。14.4 节将介绍几种备份技术。

14.3.1 充分利用MySQL服务器的自动恢复能力

MySQL 服务器的崩溃恢复能力是数据库完整性维护工作的第一道防线。其中之一（事务型存储引擎的自动恢复）在 MySQL 服务器每次启动时都会自动发生；另一种（MyISAM 自动恢复）是可选的，需要明确启用。

在启动过程中，MySQL 服务器会进行一些数据表检查工作，帮助人们解决早些时候因 MySQL 服务器或机器崩溃而导致的问题。MySQL 服务器的启动过程能自动查出和纠正很多种问题，在很多时候，你只需像往常那样重新启动你的 MySQL 服务器，就能让它替你完成许多必要的纠错和修复工作。

- 如果启用了InnoDB存储引擎，它能自动查出和纠正很多问题。比如说，对于记载在“重做日志”（redo log）里的事务（即那些已被提交但没来得及写入数据表的事务），它会重做（redo）一遍；对于记载在“撤销日志”（undo log）里的事务（即在崩溃发生时正在执行的未提交事务），它会进行回滚。这样做的结果是让你的InnoDB数据表总保持完好无损，各有关用户在崩溃发生前提交的所有事务都将毫不走样地反映在他们的InnoDB数据表的内容里。
- Falcon存储引擎（如果启用了的话）也具备类似的能力。它将根据它的serial日志去尝试进行自动恢复。

如果 InnoDB 数据表上的自动恢复工作因为一个无法恢复的问题而失败了，MySQL 服务器就会在“错误日志”里写出一条消息后退出执行。如果你打算强行启动 MySQL 服务器以尝试某种手动恢复办法，请参阅 14.7.4 节。

对于 MyISAM 数据表，MySQL 服务器提供了一种可选的数据表自动恢复能力，但你必须明确启

用。如果你启用了这一能力，MySQL 服务器就会在打开每一个 MyISAM 数据表时对它进行有关的检查。如果那个数据表在上次使用后没有被正确地关闭或是被标记为“崩溃”，服务器将检查并修复它。启用 MyISAM 数据表自动恢复能力的办法是使用`--myisam-recover=level`选项来启动 MySQL 服务器。`level` 的值是一个以逗号分隔的、由以下几个值中的一个或多个构成的列表：`BACKUP`（如果自动恢复工作需要修改某个数据表，先为它创建一个备份）、`FORCE`（强行恢复，哪怕会因此而丢失一个以上的数据行）、`QUICK`（快速恢复）或 `DEFAULT`（不进行任何其他特殊处理的普通恢复，这与把该选项设置为空的效果完全一样）。比如说，如果想在发现问题时先创建一个备份再进行强行恢复，就要使用`--myisam-recover=BACKUP, FORCE` 选项来启动 MySQL 服务器。

MyISAM 存储引擎的自动恢复能力非常有用，应该启用它，用作 MySQL 数据库系统日常维护策略的一个重要手段。如果不这样做，MySQL 管理员将只能依靠自己的经验去发现可能出现的问题并手动修复。如果在运行 MySQL 服务器时使用了`--delay-key-write` 选项或是把一些 MyISAM 数据表配置成使用“键字缓写”（delayed key write）功能，就更应该启用 MyISAM 存储引擎的自动恢复能力了。所谓“键字缓写”功能是把对索引数据的修改缓存在内存里，等到关闭数据表时候才把它们写入磁盘文件。这可以显著改善 MySQL 服务器的运行性能，但同时也意味着那些使用了“键字缓写”功能的数据表在系统发生崩溃时会有一些索引受到破坏或丢失，因而需要在下次启动 MySQL 服务器时修复它们。

14.3.2　定期进行预防性维护

除了启用 MySQL 服务器的自动恢复能力，还应该考虑制定一份预防性维护计划。这有助于自动检测故障，以便你能采取措施纠正它。有计划地检查数据表，将会减少求助于备份的可能性。在 Unix 系统上，实现这种定期检查最简便易行的办法是使用一个 `cron` 作业，通常的做法是为你用来运行 MySQL 服务器的那个账户创建一个`crontab`文件来定期执行这个`cron`作业。（`cron`作业的创建步骤见 12.5.7 节的第 3 小节。）

我们来看一个例子：`mysqlcheck` 程序可以在 MySQL 服务器运行时对 MyISAM 和 InnoDB 数据表进行检查，而你想定期调用 `mysqlcheck` 程序来进行这种检查。如果用来运行 MySQL 服务器的那个账户的用户名是 `mysql`，可以在这个用户的 `crontab` 文件里创建一个 `cron` 作业来进行这种检查。你只需在 `crontab` 文件里增加一个如下所示的设置项，但必须注意要把整个设置项输入在同一行上，还必须把 `mysqlcheck` 的完整路径写正确：

```
45 3 * * 0 /usr/local/mysql/bin/mysqlcheck
    --all-databases --check-only-changed --silent
```

如上所示的设置项告诉 `cron` 在每个星期日的凌晨 3:45 分运行 `mysqlcheck` 程序。你可以根据具体情况改变这个时间或调整其他选项。

`--all-databases` 选项将使 `mysqlcheck` 程序去检查所有数据库里的所有数据表，它可以让你的 `cron` 设置项产生最大的作用。如果想让 `mysqlcheck` 程序只检查特定的数据库或数据表，请参阅附录 F 对该程序各有关选项的描述。

`--check-only-changed` 选项告诉 `mysqlcheck` 程序跳过已在最后一次成功检查后未修改过的所有数据表，`--silent` 选项的作用是阻止输出——除非在数据表里发现错误。如果某个 `cron` 作业真的产生了输出，它通常会生成一个邮件消息。对一个用来检查数据表的 `cron` 作业来说，如果没有发现问题，也就没必要发出一封邮件。请注意，如果你的数据库有 `mysqlcheck` 程序不知道如何去检查的

存储引擎，即使你使用了--silent 选项也会收到一些诊断性输出。

说明　当一个数据表正在被检查的时候，它将不能被更新。因此，自动化的维护策略不一定适用于那些需要频繁更新的大数据表，因为在检查期间阻断对它们的更新可能会产生更严重的后果。

14.4　制作数据库备份

为了预防数据表的丢失或损坏，对数据库进行备份非常重要。在发生严重的系统崩溃时，你肯定希望自己能够以最少的数据损失把数据表恢复到崩溃之前的状态。类似地，如果某个用户不理智地发出了一条 DROP DATABASE、DROP TABLE 或 DELETE 语句，他肯定会找上门来让 MySQL 管理员帮他进行数据恢复。

数据库备份技术在需要把一个数据库复制到另一个 MySQL 服务器去的时候也很有用。把数据库转移到运行在另一台主机上的 MySQL 服务器下是最常见的，但也可以把数据转移到运行在同一台主机上的另一个 MySQL 服务器去。当你安装了 MySQL 软件的一个新版本并想用一些实际数据对它测试时，就很有可能会这样做。

备份的另一个用途是建立复制（replication）服务器。建立从服务器的前几个步骤之一，就是在某特定时刻给主服务器“拍一张照片”。这张“照片”就是对主服务器的备份，只要把它加载到从服务器里，就能让从服务器里的数据库与主服务器一模一样。此后，用户在主服务器上作出的修改将通过标准的复制协议被原样复制到从服务器上。建立复制机制的具体步骤请参阅 14.8.2 节。

我们的讨论将从备份工作的基本原则开始，这些原则可以帮你决定哪种备份技术最适用。然后，我们将对几种具体的备份技术做详细的介绍。

数据库备份按照它们的格式可以分为两大类。

- 文本格式的备份。这类备份是通过使用mysqldump程序把数据表内容写到转储文件（dump file）里而得到的，其内容主要由CREATE TABLE和INSERT两种SQL语句构成，把转储文件重新加载到MySQL服务器上就可以恢复有关的数据表。
- 二进制备份。这类备份是通过直接复制那些包含着数据表内容的文件而得到的。制作这类备份的具体办法有许多种，诸如mysqlhotcopy、cp、tar或rsync之类的程序都可以用来为数据库制作二进制备份。

每种备份方法都有它自己的优点和缺点。在挑选备份工具时需要考虑的因素包括：是否需要关停 MySQL 服务器、制作备份需要花费的时间、备份的可移植性、需要备份哪些数据库和数据表，等等。

- mysqldump程序必须与MySQL服务器配合使用，所以可以在MySQL服务器正在运行时使用它。二进制备份方法是在MySQL服务器外部进行的文件复制操作。这些方法有些需要先关停MySQL服务器。对于那些不需要关停MySQL服务器的二进制备份方法，你必须保证MySQL服务器在你复制数据表文件时不会去修改有关的数据表。
- mysqldump程序比二进制备份技术来得慢，这是因为转储操作需要通过网络在mysqldemp程序和服务器之间传输信息。二进制备份方法是在文件系统级进行文件复制，不需要通过网络传输信息。
- mysqldump程序生成的是内容为SQL语句的文本文件。这些文件很容易移植到其他的机器上，甚至可以移植到不同硬件结构的机器上，所以特别适合用来把数据库从一个服务器复制到另

一个服务器。二进制备份方法所生成的文件就不一定能移植到其他机器上了，这要取决于数据表的存储格式是否与具体的机器无关。MyISAM和InnoDB数据表通常与具体的机器无关，这两类数据表的数据表文件都可以直接复制到运行在另一台有着不同硬件结构的机器上的MySQL服务器那里去。Falcon存储引擎的日志文件和表空间文件都以一种与机器有关的格式存储，它们只在有着同样硬件特性的机器之间才是二进制可移植的。关于各种存储引擎的可移植性的讨论见2.6.1节的第11小节。

- mysqldump程序的输出只包含数据库的内容（数据表、视图、存储例程，等等），不包含没被保存在数据库里的信息，如配置文件、日志文件或复制状态文件。二进制备份的选择余地就大了，你可以在制作备份时复制任何一个文件。

无论选用哪一种备份方法，要想让今后的数据库恢复工作达到最佳效果，必须遵守以下几个原则：

- 定期对数据库进行备份。制订一个计划，并严格按照这个计划行事。
- 启用ySQL服务器的二进制日志（详见12.5节）。当需要在系统崩溃后去恢复数据库时，二进制日志能帮上大忙：用备份文件把数据库恢复到当初对它进行备份时的状态，然后运行二进制日志里的内容将重建数据表在备份以后发生的修改。这将把数据库里的数据表恢复到它们在崩溃即将发生之时的状态。
- 给备份文件起的名字既要有规律，又要有意义。诸如backup1、backup2之类的名字并没有多大的意义。等真的要用它们去进行恢复操作的时候，你得浪费些时间才能搞清楚它们里面都有哪些东西。用数据库的名字和备份日期来构造备份文件的文件名是个很不错的主意。比如说，对于在2008年1月2日为sampdb数据库制作的备份，可以把相应的备份文件命名为sampdb-2008-01-02。如果同时运行着多个MySQL服务器，还应该加上一个用来区分各服务器的标识符。
- 不要把备份文件和数据库放在同一个文件系统上。首先，这可以避免备份文件挤占MySQL数据目录的文件系统。其次，如果用来存放你备份文件的文件系统位于另一个驱动器上，这还能降低因硬盘故障而导致的损失程度——除非两个驱动器同时发生故障，你的MySQL数据目录和备份文件总会有一个能“幸存”下来。
- 定期使用文件系统的备份功能来备份你的数据库备份文件。如果系统崩溃不仅让你失去了MySQL数据目录，还殃及你用来存放数据库备份文件的硬盘，你就真的麻烦了。别忘了把日志文件也备份下来。
- 定期对备份文件进行失效处理以防止它们填满你的硬盘。办法之一是使用文件轮转技术。12.5.7节介绍了几种针对日志文件的这类技术，同样的原则完全适用于对备份文件进行的失效处理。

接下来几节将介绍几种具体的备份方法。如果你正使用复制机制，阅读 14.8.4 节，其中介绍了一种可以在完全不打扰主服务器的情况下进行数据库备份的方法。

14.4.1　用mysqldump程序制作文本备份

在使用 mysqldump 程序去创建文本转储文件时，备份文件将默认地写为 SQL 格式，它由一系列 CREATE TABLE 和 INSERT 语句组成，CREATE TABLE 语句负责创建被备份的数据表，INSERT 语句则包含有该数据表中各数据行的数据。等以后重建数据库的时候，只要把 mysqldump 的转储文件用作 mysql 程序的输入，就能把它们重新加载到 MySQL 数据库系统里去。下面的命令可以用来转储和重

新加载一个给定的数据表（以 sampdb.member 数据表为例）：

```
% mysqldump sampdb member > member.sql
% mysql sampdb < member.sql
```

千万不要用 mysqlimport 程序去重新加载 mysqldump 程序的 SQL 格式的输出！mysqlimport 程序只能读取数据行，不能用来读取 SQL 语句。

下面的命令将把所有数据库里的所有数据表备份到同一个文件里：

```
% mysqldump --all-databases > /archive/mysql/dump-all.2008-01-02
```

不过，如果数据量非常大的话，如此制作的转储文件往往相当巨大。可以按如下所示的命令将每个数据库转储到它自己的文件：

```
% mysqldump mysql > /archive/mysql/mysql.2008-01-02
% mysqldump sampdb > /archive/mysql/sampdb.2008-01-02
% ...
```

mysqldump 输出文件的开头部分看起来是下面这样：

```
-- MySQL dump 10.11
--
-- Host: localhost    Database: sampdb
-- ------------------------------------------------------
-- Server version       5.0.54-log

 ... several SET statements ...

--
-- Table structure for table `absence`
--

DROP TABLE IF EXISTS `absence`;
CREATE TABLE `absence` (
  `student_id` int(10) unsigned NOT NULL,
  `date` date NOT NULL,
  PRIMARY KEY  (`student_id`,`date`),
  CONSTRAINT `absence_ibfk_1` FOREIGN KEY (`student_id`)
  REFERENCES `student` (`student_id`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
--
-- Dumping data for table `absence`
--
LOCK TABLES `absence` WRITE;
/*!40000 ALTER TABLE `absence` DISABLE KEYS */;
INSERT INTO `absence` VALUES (3,'2008-09-03'),(5,'2008-09-03'),
(10,'2008-09-06'),(10,'2008-09-09'),(17,'2008-09-07'),(20,'2008-09-07');
/*!40000 ALTER TABLE `absence` ENABLE KEYS */;
UNLOCK TABLES;
...
```

这种文件的后续内容是更多的 SQL 语句，如 CREATE TABLE 和 INSERT 语句。

转储文件的尺寸通常都比较大，所以你肯定想把它们弄小一些。mysqldump 程序有一个--opt 选项，它使另外几个选项可以对转储过程进行优化以生成尺寸较小的输出。因为那些输出在你重新加载转储文件时可以更快地被处理，所以这些选项对数据库的恢复工作也有优化效果。比如说，--opt 选

项的效果之一是让 mysqldump 程序生成的 INSERT 语句可以一次插入多个数据行。与每次只插入一个数据行的 INSERT 语句相比，一次插入多个数据行的 INSERT 语句既可以减少空间占用量，又可以加快重新加载数据表的速度。

从 MySQL 4.1 版开始，--opt 选项是默认启用的，不需要再明确地给出。如果想禁用它，请使用 --skip-opt 选项。

减少转储文件长度的另一个办法是对它进行压缩。在 Windows 系统上，可以选用 WinZip 或类似的程序来压缩转储并生成一个 Zip 格式的文件。在 Unix 系统上，可以选用 gzip 或 bzip2 压缩工具。我们甚至可以在命令行上使用一个管道在制作备份的同时对它进行压缩：

```
% mysqldump sampdb | gzip > /archive/mysql/sampdb.2008-01-02.gz
% mysqldump sampdb | bzip2 > /archive/mysql/sampdb.2008-01-02.bz2
```

如果觉得尺寸过大的转储文件难以管理，还可以分别转储各个数据表的内容，在 mysqldump 命令行上的数据库名后面给出相应的数据表的名字即可。mysqldump 程序将只转储给定数据库里的给定数据表，而不是给定数据库里的全体数据表。如此得到的备份文件尺寸会小很多，管理起来自然就容易多了。下面这个例子将把 sampdb 数据库里的几个数据表分别转储到不同的文件：

```
% mysqldump sampdb member president > hist-league.sql
% mysqldump sampdb student score grade_event absence > gradebook.sql
```

mysqldump 程序有许多选项。下面是这些选项中比较常用的。

- 人们通常只在mysqldump命令行上给出一个数据库名，在它后面再选择性地给出几个数据表名。如果你想一次转储多个数据库，就要使用--databases选项。mysqldump将把你在命令行上给出的名字全部解释为数据库名，并依次转储这些数据库里的所有数据表。如果你想转储某服务器里的所有数据库，可以使用--all-databases选项，此时不用给出任何数据库名或数据表名参数。--databases或--all-databases选项都会在每个数据库输出前插入必要的CREATE DATABASE IF NOT EXISTS和USE语句。

  如果打算把转储文件加载到另一个服务器去，请慎重使用--all-databases 选项：转储文件里将包括 mysql 数据库里的各个权限表，而你未必真想替换另一个服务器的权限表。
- 在默认情况下，mysqldump程序将同时备份数据表的结构（CREATE TABLE语句）和数据表的内容（INSERT语句）。如果只想转储其中之一，请使用--no-create-info或--no-data选项。
- 如前所述，--opt选项可以对转储过程进行优化。--opt选项将启用其他几个能加快数据转储速度的选项。此外，转储文件里的信息会被写成将在数据加载工作中得到更快处理的格式。--opt选项是默认启用的，不需要再明确地给出。但如果你真的不需要优化转储，可以使用--skip-opt选项禁用之。

  使用--opt 选项进行备份可能是最常见的做法了，因为它可以加快备份速度。但使用--opt 选项是有代价的：--opt 选项优化的是备份操作，不是其他客户对数据库的访问操作。--opt 选项会把你正在转储的数据表全都锁定上，不让任何人对它们中的任何一个进行修改。这种做法对数据库用户们的影响很容易看到：只要在数据库平时最忙的时候开始制作备份，用不了多一会儿，肯定会有人打电话来问你出了什么事。（如果你能忍住不问我是怎么知道会发生这种事情的，我会非常感谢。）

  如果制作备份文件是为了用它们去定期更新另一个数据库（比如说，另一个服务器上的某个数据库）的内容，--opt 选项就会很有用。这是因为--opt 选项将自动启用--add-drop-table

选项，后者将使 mysqldump 程序在备份文件里的每条 CREATE TABLE 语句之前加上一条 DROP TABLE IF EXISTS 语句，这两条语句中的数据表名字完全一样。这样，当你把备份文件加载到第二个数据库时，如果数据表已经存在，也不会发生错误。如果你想对另一个 MySQL 服务器进行测试却又不想把它设置为复制机制中的从服务器，就可以利用这个技巧定期地把你在第一个 MySQL 服务器上制作的数据库备份拿到第二个 MySQL 服务器上把数据加载进去。

- --opt选项的效果之一是它将启用--extended-insert选项，后者将使mysqldump程序生成可以一次插入多个数据行的INSERT语句。这种INSERT语句往往会降低备份文件的可读性。如果想让mysqldump程序生成的INSERT语句每条只插入一个数据行，请使用--skip-extended-insert选项。
- --flush-logs和--lock-all-tables选项的组合能在数据库检查工作中帮上你的忙。--lock-all-tables选项用来申请一个全局级读操作锁，--flush-logs选项用来关闭再重新打开日志文件。如果启用了二进制日志功能，刷新日志将创建一个新的二进制日志文件来记载此后发生的数据修改。这样就能把你的日志同步到你进行备份时的时间。（副作用是，在备份期间锁定所有的数据表的做法对客户不太友好。）

  如果打算用--flush-logs 选项把日志同步到你进行备份的时间，最好转储整个数据库。在恢复操作期间，按数据库提取日志内容的做法是很常见的。如果你转储的是各个数据表，让日志检查点与备份文件保持同步就比较困难。（mysqldump 程序没有提供按数据表提取日志内容的选项，只能依靠自己去提取它们。）
- 在转储InnoDB或Falcon数据表的时候，最好是用--single-transaction选项把转储操作放在一个事务里执行，这可以确保得到一个稳定的备份。
- 如果数据库包含着存储例程、触发器和事件，可以用--routines、--triggers和--events选项明确地把它们包括到转储输出里，这三个选项分别始见于MySQL 5.0.13、5.0.11和5.1.8版本。这几个选项还各有一个--skip形式（例如--skip-triggers）用来禁止转储指定的对象。在默认的情况下，只有触发器将被包括在转储输出里，存储例程和事件不进入转储。

mysqldump 程序还有很多其他的选项；详见附录 F。

14.4.2　制作二进制数据库备份

要备份一个数据库或数据表、但又不使用 mysqldump 的另一种方法是直接复制数据表文件。典型做法是使用常规的文件系统工具（如 cp、tar 或 rsync）来完成这项工作，或者使用专门为此开发的工具（比如 mysqlhotcopy 或 InnoDB Hot Backup）。在使用直接复制法制作备份的时候，有两个要点必须注意。

- 一定要确保没有人在使用那个数据表。如果你这边在复制、MySQL服务器那边在修改，复制出来的数据表就没有实际价值。保证复制完整性的最好方法是关停MySQL服务器，复制有关文件，然后重新启动MySQL服务器。事实上，有几种二进制备份方法要求你必须关停MySQL服务器。如果不想关停MySQL服务器（并且备份方法也没有这么要求），就应该使用某种只读锁定机制以防止MySQL服务器在你复制数据表文件的同时去修改相应的数据表。请参阅14.2节。
- 在以直接复制法备份某个数据表时，一定要把恢复这个数据表所需要的文件全都复制下来。直接复制法最适用于那些把每个数据表分别表示为数据目录里的一组文件的存储引擎，如

MyISAM等。如果想备份一个MyISAM数据表，只需复制它的.frm、.MYD和.MYI文件就行了。对于InnoDB这样的存储引擎，事情就麻烦得多了：除必须复制.frm文件以外，还必须复制所有的表空间文件和InnoDB日志文件。

在制作二进制备份时，一定要注意符号链接（比如 MySQL 数据目录里指向各数据库的符号链接、指向 MyISAM 数据或索引文件的符号链接）可能带来的问题：你使用的文件复制技术可能只复制那些符号链接本身，而不是它们所指向的数据。

1. 制作一个完整的二进制备份

完整的二进制备份至少应该包括存放着数据表内容的所有文件和供特定存储引擎使用的各种日志文件。为保险起见，还应该把二进制日志文件也复制下来。如果是在复制机制中的从服务器上制作备份，还应该把中继日志（relay log）文件以及 master.info 和 relay-log.info 文件复制下来。还有，别忘了到从服务器的临时文件子目录里查看一下那里有没有名字是 SQL_LOAD-*xxx* 形式的文件。如果有，把这些也备份下来，它们是 LOAD DATA 语句所需要的。这些文件应该在--slave-load-tmpdir 选项给出的子目录里；如果这个选项没给出，它的默认值等于 tmpdir 系统变量的值。为了在备份时容易找到这些文件，应该提前在从服务器上为--slave-load-tmpdir 选项创建一个专用的子目录并在启动从服务器时用上这个选项。

要想把上面提到的这些文件全都正确地复制下来，必须关停 MySQL 器并保证关停过程没有错误，这是为了确保各存储引擎关闭了它们的日志文件，服务器也关闭了它正在写的任何其他日志。

说了这么多，你可能觉得制作一个完整的二进制备份很麻烦，但实际往往没那么复杂。比如说，所有的数据库子目录都在数据目录下，日志和信息文件也都默认地在那里创建。此时，可以通过关停服务器并复制整个数据目录的办法来制作备份。比如说，如果想把 MySQL 数据目录整个地备份为/archive/mysql 子目录中的一个 tar 压缩文件，用如下所示的命令切换到 MySQL 数据目录并进行备份即可：

```
% cd /usr/local/mysql/data
% tar czf /archive/mysql/backup-all-2008-04-11.tar.gz .
```

2. 制作一个部分的二进制备份

通过复制文件的办法来制作一个部分的二进制备份和制作一个完整的备份很相似，只不过需要复制的文件是全部文件的一个子集而已。比如说，如果打算对目录/usr/local/mysql/data 里的 mydb 数据库进行备份，并把备份保存到/archive/mysql 子目录下，先关停 MySQL 服务器，然后执行下面这些命令：

```
% cd /usr/local/mysql/data
% cp -r mydb /archive/mysql
```

执行完这些命令之后，/archive/mysql/mydb 子目录将包含 mydb 数据库的一份副本。各个MyISAM 数据表可以用如下所示的语句进行备份。（请注意：一定要提前把/archive/mysql/mydb 子目录创建出来！）

```
% cd /usr/local/mysql/data/mydb
% cp tbl1.* /archive/mysql/mydb
% cp tbl2.* /archive/mysql/mydb
...
```

完成备份工作后，重新启动 MySQL 服务器。

在某些场合，如果使用了一种读锁定机制来锁定你想复制的数据表，在没有关停 MySQL 服务器的情况下也可以制作一个部分备份。比如说，只包含 MyISAM 数据表的数据库就可以用这个办法来备份。如果在前几个例子里用的 mydb 数据库恰好是这种情况的话，就可以按以下步骤来进行备份：给数据表加上读操作锁并刷新，执行备份命令，备份完成后释放数据表上的读操作锁。

3. 用mysqlhotcopy工具制作备份

mysqlhotcopy 工具是一个能帮我们制作数据库备份的 Perl DBI 脚本。它名字中的“hot”指的是备份工作将在 MySQL 服务器保持运行时进行，我们不必关停它。

mysqlhotcopy 脚本的优点主要有以下几点。

- 比mysqldump程序更快。这是因为它将直接复制数据表文件，不像mysqldump程序那样需要通过MySQL服务器请求数据表。（这也意味着你必须在服务器主机上运行这个脚本，它不能用来对远程MySQL服务器进行备份。）
- 使用方便。这个脚本能替你完成必要的锁定工作，阻止MySQL服务器对你正复制的数据表进行修改。它是通过使用14.2.1节的第1小节描述的内部锁定机制做到这一点的。
- 它能刷新日志，让备份文件和日志文件的检查点保持同步。在以后进行恢复的时候，如此制作出来的备份更容易使用。

mysqlhotcopy 脚本的局限性主要有以下几点。

- 它只能在MySQL服务器正在运行时使用，因为它必须通过MySQL服务器去续锁定它将要复制的数据表。因为它直接访问数据表文件，所以它必须在服务器主机上运行。
- 它只能用来备份MyISAM和ARCHIVE数据表。
- 它只能在Unix和NetWare系统上使用，不能在Windows系统上使用。

在下面的几个例子里，我们需要假设将被备份的数据库只包含 MyISAM 或 ARCHIVE 数据表。

mysqlhotcopy 脚本有好几种调用方法。假设你想复制 mydb 数据库，下面的命令将在 MySQL 数据目录里创建一个名为 mydb_copy 的子目录并把 mydb 数据库子目录里的文件复制到那里去：

```
% mysqlhotcopy mydb
```

这条命令本身没有任何问题，但由于新数据目录里的备份子目录在 MySQL 服务器看来将是一个可以访问的数据库，所以最好不要把数据库的备份存放在数据目录里。这一事实很容易验证：只要在执行完上面那条命令后发出一条 SHOW DATABASES 语句，就会在输出里同时看到 mydb 和 mydb_copy。而这意味着其他客户可以连接到这个 MySQL 服务器去修改你放在备份子目录里的数据表。

如果想把 mydb 数据库复制到某个特定的子目录下，请在数据库名的后面给出那个子目录的路径名。比如说，下面这条命令将把 mydb 数据库复制到一个名为/achive/mysql/mydb 的子目录里去：

```
% mysqlhotcopy mydb /archive/mysql
```

如果你想了解 mysqlhotcopy 脚本各子命令的用法，请在 mysqlhotcopy 命令行上增加一个 --dryrun 或-n 选项。这个选项将使 mysqlhotcopy 运行在“不执行”状态，即只显示有关的命令但不实际执行它们。

14.4.3 备份InnoDB或Faclcon数据表

用于 InnoDB 和 Falcon 等事务型存储引擎的数据表也可以用 mysqldump 程序来转储，就像任何其他类型的数据表一样。--single-transaction 选项对事务型存储引擎来说非常有用，它将使

mysqldump 程序把有关数据表当做事务的一部分来转储。对于 InnoDB 和 Falcon 存储引擎，这将确保数据表在转储期间不会被修改，使你得到一个稳定可用的备份。（但这个选项无法保证为 MySQL Cluster 提供一个稳定可用的备份。）

我们还可以使用 InnoDB Hot Backup 工具来制作 InnoDB 数据表的二进制备份，这个工具可以从 Innobase 公司的网站下载。InnoDB Hot Backup 是一个专门用来在 MySQL 服务器正在运行时制作 InnoDB 备份的商业化工具。关于这个工具的详细资料请访问 http://innodb.com。

如果想使用直接复制法来制作二进制 InnoDB 备份，必须注意以下几个特殊要求。

- InnoDB存储引擎有它自己的日志文件用于事务管理，这些日志文件在服务器运行时都是可用的。因此，要想制作二进制InnoDB备份，必须先关停MySQL服务器，而且服务器的关停过程必须没有任何错误，这是为了确保InnoDB存储引擎有机会完成尚未完成的事务并正确地关闭它的日志。
- 在为InnoDB数据表制作二进制备份的时候，必须把以下文件全部复制下来。
  - 共享表空间文件。
  - 每个数据表的.frm文件。
  - 每个数据表的.ibd文件（如果InnoDB存储引擎被配置成给每个数据表分别创建一个表空间文件的话）。
  - InnoDB日志文件。
  - 用来配置共享表空间的选项文件。（为选项文件制作一份副本的目的是：即使当前的选项文件丢失了，你也能对共享表空间重新进行初始化。）

为 Falcon 数据表制作二进制备份的步骤与 InnoDB 数据表的情况相似，只不过需要复制的是 Falcon 存储引擎的表空间文件和日志文件（以.fts、.fl1 和.fl2 为扩展名的文件）而已。

14.5　把数据库复制到另一个服务器

数据库备份可以用来把某给定数据库从一个 MySQL 服务器复制到另一个 MySQL 服务器。本节将介绍几种完成这种数据库迁移的办法。为便于讨论，我将假设我们的目标是把一个数据库从本地主机上的服务器迁移到远程主机 boa.snake.net 上的另一个 MySQL 服务器。不过，这两个服务器完全可以运行在同一台主机上。此外，虽然我们将要讨论的是如何复制整个数据库，但同样的技巧也用来复制数据表。

我们将介绍两种把数据库复制到另一个服务器的办法。第一个办法是先把数据库备份为一个或一组文件，然后把那些文件复制到第二台服务器主机并把它们加载到第二个 MySQL 服务器里。第二个办法是在第一个服务器里备份数据库的同时通过网络把它直接转储到另一个服务器去，这么做的好处是无需使用任何中间文件。

14.5.1　使用一个备份文件来复制数据库

使用一个文本备份文件来复制数据库的基本思路是：先用 mysqldump 程序创建出备份文件，然后把备份文件复制到第二台服务器主机，把备份文件加载到第二台主机上的 MySQL 服务器里。下面的例子演示了如何复制 sampdb 数据库的步骤。

(1) 创建一个转储文件：

```
% mysqldump --databases sampdb > sampdb.sql
```

--databases 选项将使 mysqldump 程序在备份文件里为 sampdb 数据库增加 CREATE DATABASE IF EXISTS 和 USE 语句。当你在远程主机上加载转储文件时，它将自动创建 sampdb 数据库并把它用做默认数据库，然后把转储数据表加载到数据库里。

(2) 把转储文件复制到远程主机。下面这条命令将使用 scp 程序把文件复制到 boa.snake.net 主机上的/tmp 子目录里：

```
% scp sampdb.sql boa.snake.net:/tmp
```

(3) 登录进入远程主机，把转储文件加载到该主机上的 MySQL 服务器里：

```
% mysql < /tmp/sampdb.sql
```

另一个办法是使用二进制备份技术：把数据库文件（请注意，不是转储文件）从一台主机复制到另一台主机。假设 mydb 数据库里的数据表全都是 MyISAM 数据表。此时，数据表信息完全包含在 mydb 数据库子目录中的文件里。如果本地数据目录是/usr/local/mysql/data，远程主机 boa.snake.net 上的 MySQL 数据目录是/var/mysql/data。下面的命令将把 mydb 数据库目录复制到那台主机上：

```
% cd /usr/local/mysql/data
% scp -r mydb boa.snake.net:/var/mysql/data
```

要想用这个办法把数据库文件复制到另一台主机，必须满足以下要求。

- 两台机器必须有同样的硬件结构，或者你将复制的数据表都是可移植的数据表类型。要不然，目标主机上的数据表就可能出现非常奇怪的内容。
- 你在两台主机上都必须阻止MySQL服务器在你复制文件的同时去修改它们。最安全的做法是先关停那两个MySQL服务器，等复制完数据库文件再重新启动它们。

14.5.2 把数据库从一个服务器复制到另一个

上一小节介绍的办法需要先用 mysqldump 程序创建一个转储文件并把它复制到目标服务器主机去。如果把 mysqldump 程序的输出通过网络直接写到另一个服务器去，就不需要使用转储文件作为媒介了。比如说，下面这条命令将把 sampdb 数据库从本地主机复制到远程主机 boa.snake.net 上的 MySQL 服务器去：

```
% mysqldump --databases sampdb | mysql -h boa.snake.net
```

在上面这条命令里，mysql 程序将读取转储输出、连接到 boa.snake.net 主机上的 MySQL 服务器，并把备份出来的数据转储到那个服务器里。

如果你从本地主机无法访问远程 MySQL 服务器但能够通过登录远程主机的办法去访问它，你就可以像下面这样通过 ssh 来远程调用 mysql 程序：

```
% mysqldump --databases sampdb | ssh boa.snake.net mysql
```

如果是通过一个慢速网络把数据库复制到另一台机器，--compress 选项可以改善性能，因为它能减少在网络上传输的数据量：

```
% mysqldump --databases sampdb | mysql --compress -h boa.snake.net sampdb
```

请注意，--compress 选项是在与远程主机进行通信的命令行（例子中的 mysql）里给出的，而不是在与本地主机进行通信的命令行里给出的。这里所说的“压缩”指的是经网络传输的数据量，它

不会导致在目标数据库里创建出被压缩过的数据表来。

14.6　数据表的检查和修复

会让数据库受损的原因有很多，受损的轻重程度也有很大的变化。如果运气够好，可能只有一两个数据表受到了轻微的破坏（比如说，你的机器因为停电而关停）。对于这种情况，MySQL 服务器往往能在恢复正常运行时修复受损的数据表。如果运气不够好，没准儿就得更换整个数据目录（比如说，如果硬盘发生故障并破坏了你的数据目录）。其他场合也可能需要进行数据库恢复，比如说当用户错误地丢弃了某个数据库或数据表，或者删除了数据表的内容。这些不幸事件的原因无论是什么，你都必须执行恢复。

本节将介绍一些数据表的检查和维护方法，可以用它们来解决很多不那么严重的问题。如果怀疑某个数据表受到了破坏，请先对它进行检查。检查数据表是否出了问题。如果没查出任何问题，事情也就到此为止；否则，按以下步骤修复。

- 先试试比较快速但不那么全面的修复方法，看它能否解决问题。
- 如果那个方法都不管用，试试那些更全面（但更慢）的方法，直到问题解决或是你再也找不到更好的方法为止。

在实际工作中，绝大多数问题都很容易解决，用不着求助于更全面但更慢的修复方法。

万一数据表或数据库已丢失或无法修复，就需要使用数据库备份和二进制日志来恢复它们。具体做法和步骤参见 14.7 节。

下面是检查和修复 MyISAM 和 InnoDB 数据表的几种常用办法；随后的几节将对它们做详细介绍。

检查和修复 MyISAM 数据表的常用办法包括以下几种。

- 发出CHECK TABLE和REPAIR TABLE语句。也可以使用mysqlcheck程序，它将替你连接服务器并发出那些语句。
- 使用myisamchk程序，它将直接在数据表文件上操作。

正如本章前面提到的那样，如果在让 MySQL 服务器进行这项工作和运行外部工具程序之间选择的话，则应该让 MySQL 服务器完成这项工作比较容易。这样你不必考虑使用锁定机制来协调数据表访问冲突的问题。使用 CHECK TABLE 和 REPAIR TABLE 语句（或 mysqlcheck 程序）就有这个好处。如果使用 myisamchk 程序，你就必须保证 MySQL 服务器不会在你正在修复数据表的时候使用它们。不过，你也可能决定使用 myisamchk 程序，主要理由如下。

- mysqlcheck程序可以在MySQL服务器关停时使用。CREAT TABLE和REPAIR TABLE语句只能在MySQL服务器正在运行时使用。
- 你可以加大myisamchk程序使用的缓冲区，从而加快数据表的检查和修复速度。这在处理非常巨大的数据表时非常有帮助。

CHECK TABLE 语句和 mysqlcheck 程序也可以用来检查 InnoDB 数据表。如果在一个 InnoDB 数据表里发现问题，先用 mysqldump 程序转储它，然后删除原来的数据表，最后通过重新加载转储文件的办法重建。下面几条命令给出了检查、删除和重新加载 sampdb 数据库里的 absence 数据表的步骤：

```
% mysqlcheck sampdb absence
% mysqldump sampdb absence > absence.sql
% mysql sampdb < absence.sql
```

14.6.1 用服务器检查和修复数据表

要让服务器检查和修复数据表，就要使用 CHECK TABLE 和 REPAIR TABLE 语句或 mysqlcheck 程序来进行检查和修复。

1. 用CHECK TABLE语句检查数据表

CHECK TABLE 语句为我们提供了一个使用 MySQL 服务器的数据表检查能力的接口。这条语句可以用来检查 MyISAM 和 InnoDB 数据表，还可以用来检查视图（从 MySQL 5.0.2 版本开始）、ARCHIVE 数据表（从 MySQL 5.0.16 版本开始）和 CSV 数据表（从 MySQL 5.1.9 版本开始）。

要使用 CHECK TABLE 语句，先列出一组数据表名称，然后给出一个或多个可选的限定符来表明你想进行何种类型的检查。比如说，下面这条语句将对 3 个数据表进行中级检查，如果它们没被正确地关闭的话。

```
CHECK TABLE tbl1, tbl2, tbl3 FAST MEDIUM;
```

CHECK TABLE 语句的所有检查选项可以在附录 E 里查到。那些选项全都可以用于 MyISAM 数据表，但不一定能用于其他的存储引擎。

在某些情况下，CHECK TABLE 语句有可能实际改变被检查的数据表。比如说，如果某个数据表被标记为“崩溃”或是没被正确地关闭但检查结果是没有发现任何问题，CHECK TABLE 语句将把它重新标记为“完好”。这种改变只涉及一个内部标志。

2. 用REPAIR TABLE语句修复数据表

REPAIR TABLE 语句为我们提供了一个使用 MySQL 服务器的数据表修复能力的接口。这条语句可以用来修复 MyISAM 和 ARCHIVE 数据表，还可以用来修复 CSV 数据表（从 MySQL 5.1.19 版本开始）。

要使用 REPAIR TABLE 语句很容易使用，先列出一组数据表名称，然后给出一个或多个可选的限定符来表明你想进行何种类型的修复。比如说，下面这条语句将对 3 个数据表进行快速修复：

```
REPAIR TABLE tbl1, tbl2, tbl3 QUICK;
```

REPAIR TABLE 语句的所有修复选项可以在附录 E 里查到。那些选项全都可以用于 MyISAM 数据表，但不一定能用于其他的存储引擎。

14.6.2 用mysqlcheck程序检查和修复数据表

mysqlcheck 程序是 CHECK TABLE 和 REPAIR TABLE 语句的命令行界面。这个程序将连接到服务器并根据你给出的选项替你发出相应的语句。也正是因为如此，myisamcheck 程序能够检查和修复的数据表与 CHECK TABLE 和 REPAIR TABLE 语句能够检查/修复的数据表是相对应的。

这个工具的典型用法是在程序名 myisamcheck 的后面给出一个数据库的名字，然后是一个或多个可选的数据表的名字。如果只给出了一个数据库的名字，myisamcheck 程序将检查该数据库里的所有数据表：

```
% mysqlcheck sampdb
```

如果在数据库名的后面还给出了一些数据表的名字，mysqlcheck 程序将只检查那些数据表：

```
% mysqlcheck sampdb president member
```

如果给出了--databases 选项，随后的参数将被解释为数据库名，myisamcheck 程序将依次检

查那些数据库里的所有数据表：

```
% mysqlcheck --databases sampdb test
```

如果给出了--all-databases 选项，mysqlcheck 程序将检查所有数据库里的所有表数据表，用不着再给出任何数据库或数据表的名字作为参数：

```
% mysqlcheck --all-databases
```

与直接发出 CHECK TABLE 和 REPAIR TABLE 语句相比，mysqlcheck 程序更容易使用，因为那些语句需要你明确地给出你想检查或修复的数据表的名字。如果需要对某给定数据库里的所有数据表进行检查，使用 mysqlcheck 程序要更容易一些：它将替你查出该数据库都包含有哪些数据表，并正确地构造出用来对每个数据表进行检查的语句。

在默认的情况下，mysqlcheck 程序将对数据表进行中级检查，但我们可以使用有关选项明确地选择我们想让它进行何种类型的检查。下表列出了一些 mysqlcheck 程序选项和与之相对应的 CHECK TABLE 语句选项。（请注意：CHECK TABLE 语句的所有选项都适用于 MyISAM 数据表，但不一定适用于其他的存储引擎。）

mysqlcheck程序选项	**CHECK TABLE**语句选项
--check-only-changed	CHANGED
--extended	EXTENDED
--fast	FAST
--medium-check	MEDIUM
--quick	QUICK

mysqlcheck 程序还可以完成数据表修复操作，但仅限于 MyISAM 数据表。下表列出了一些 mysqlcheck 程序选项和与之相对应的 REPAIR TABLE 语句选项。（请注意：REPAIR TABLE 语句的所有选项都适用于 MyISAM 数据表，但不一定适用于其他的存储引擎。）

mysqlcheck程序选项	**REPAIR TABLE**语句选项
--repair	无选项（执行一个标准的修复操作）
--repair --extended	EXTENDED
--repair -quick	QUICK
--repair --use-frm	USE_FRM

14.6.3　用**myisamchk**程序检查和修复数据表

myisamchk 工具程序可以检查和修复 MyISAM 数据表。myisamchk 程序通过直接访问数据表文件来完成其工作，所以在开始使用这个工具之前最好先关停 MySQL 服务器以防止它和 myisamchk 程序同时访问同一个数据表文件。如果想让 MySQL 服务器保持运行，就必须按照 14.2.1 节给出的步骤使用适当的锁定机制来防止 MySQL 服务器在你使用 myisamchk 程序检查或修复某个数据表的同时使用该数据表。下面的讨论将假设你已经关停了 MySQL 服务器或是使用了适当的锁定机制。

myisamchk 程序不会对数据表的存放位置做出任何假设，你必须在运行这个程序时明确地给出你打算检查或修复的数据表文件的路径名。显然，先切换到那些数据表所在的子目录再进行操作是最方便的做法。一般来说，在调用 myisamchk 程序之前应该先进入有关的数据库子目录，再告诉它你想检查或修复哪个数据表并给出选项来表明你想进行的操作：

```
% myisamchk options tbl_name...
```

tbl_name 参数既可以是一个数据表的名字，也可以是一个数据表的索引文件的名字。下面两条命令是等价的：

```
% myisamchk member
% myisamchk member.MYI
```

如果需要给出某数据库子目录里的所有相关索引文件的名字，可以使用一个文件名模板（当然，这么做的前提是你的命令解释器必须能够理解并正确处理文件名通配符）：

```
% myisamchk options *.MYI
```

如果不想对原始数据表文件进行一个 myisamchk 操作，可以先把它们复制到另一个子目录里，然后切换到那个子目录对那些副本进行处理。

1. 用myisamchk程序检查数据表

myisamchk 程序可以对数据表进行多种检查，它们之间的主要区别在于对数据表的检查是否全面。如果只想进行“普通”检查，可以使用下面两条命令中的任何一个：

```
% myisamchk tbl_name
% myisamchk --check tbl_name
```

如果不带任何选项，myisamchk 程序的默认行为将是进行--check 检查，所以上面两条命令是等价的。

myisamchk 程序默认进行的“普通”检查已足以查出绝大多数问题。如果它报告说没有错误、但你仍怀疑有问题（比如说，当你觉得查询结果不正确的时候），你可以用--medium-check 选项再做一次“中级”检查。这需要花费更多的时间，但可以检查得更全面和更彻底。作为最后一招，还可以用--extend-check 选项进行“高级”检查。这将花费非常多的时间，但最全面彻底：对于数据表的数据文件里的每一个数据行，索引文件里每一个与之相关的索引项的键都将受到检查以确保它确实指向正确的数据行。

如果用--extend-check 选项进行的高级检查没报告任何错误，就可以确认数据表是完好的。如果在使用数据表时仍遇到了问题，应该从其他方面找原因。请认真检查每一条执行出错的语句本身是否正确。如果能够确认问题出在 MySQL 服务器身上，可以考虑填写一份 bug 报告或是把 MySQL 软件升级到一个更高的版本。

如果 myisamchk 程序报告数据表有错误，应该尽量修复它。

2. 用myisamchk程序修复数据表

如果需要使用 myisamchk 程序修复数据表，请按以下步骤进行。这里的原则是：先尝试速度较快但不那么全面彻底的修复办法，如果不能解决问题，再尝试速度较慢但更加全面彻底的修复办法。在开始修复数据表之前，应该先备份数据表文件以防万一。人们在修复数据表时都会很谨慎，不容易犯错误，但万一出了问题，可以用刚才制作的备份恢复数据表（虽然它有问题），然后再去尝试另一种修复方法。

说明 如果数据表包含 FULLTEXT 索引，在修复它时可能需要用到 myisamchk 程序的其他选项。这方面的细节请参阅附录 F 在介绍 myisamchk 程序时给出的关于 FULLTEXT 索引的说明。

---

(1) 首先尝试使用--recover 选项来修复数据表，还可以加上--quick 选项，只根据索引文件的

内容来修复数据表，不涉及数据文件：

```
% myisamchk --recover --quick tbl_name
```

(2) 如果问题没有解决，去掉--quick 选项后再执行一次这个命令。此时，myisamchk 程序还可以对数据文件做必要的修改：

```
% myisamchk --recover tbl_name
```

(3) 如果问题还是没有解决，试试--safe-recover 修复模式。这比其他修复模式的速度都要慢，但能修复一些--recover 模式无法修复的问题：

```
% myisamchk --safe-recover tbl_name
```

执行这些命令时，myisamchk 程序可能会在显示一条“Can't create new temp file: *file_name*”（无法创建新的临时文件）出错消息后退出运行，而导致这条出错消息的常见原因，是上次修复失败时创建的临时文件还留在那里没有被删除。要想强行删除以前留下的临时文件，给刚才的命令加上--force 选项后再执行一遍。

如果上述修复步骤未能修复数据表，其原因往往是数据表的索引文件已经丢失或无法修复，也有可能（很少见）是因为数据表的.frm 格式文件已经丢失。如果遇到这两种情况，必须更换受影响的文件，然后再次尝试上述修复步骤。

下面是为数据表 t 重新生成索引文件的步骤，其前提是该数据表的格式文件 t.frm 完好无损。

(1) 切换到受损数据表所在的数据库子目录。

(2) 把数据表的数据文件 t.MYD 移动到一个安全的地方。

(3) 执行下面这条语句，让 mysql 程序重新创建一个新的空白数据表：

```
mysql> TRUNCATE TABLE t;
```

TRUNCATE TABLE 语句将使用数据表格式文件 t.frm 重新生成一个新的数据文件和一个新的索引文件。

(4) 退出 mysql 程序，把原来的数据文件移回当前数据库子目录，用它替换掉你刚创建的空白数据文件。这将导致数据文件和索引文件不匹配，但索引文件现在有了一个合法的内部结构，MySQL 服务器可以根据数据文件和数据表格式文件的内容解释并重建索引。

(5) 按照本节开头部分给出的步骤再次尝试修复数据表。

如果没有受损数据表的.frm 格式文件，将需要从数据库备份里先把那个.frm 文件恢复出来，再按照本节开头部分给出的步骤去修复该数据表。如果那个.frm 文件已经丢失并且没有备份，但你还记得当初创建该数据表时使用的 CREATE TABLE 语句，可以按照以下步骤来修复它：

(1) 切换到受损数据表所在的数据库子目录。

(2) 把数据表的数据文件 t.MYD 移动到一个安全的地方。如果要使用索引文件 t.MYI，应该把它也移动到安全的地方。

(3) 运行 mysql 程序并发出 CREATE TABLE 语句重新创建数据表。这将创建出新的.frm、.MYD 和.MYI 文件。

(4) 退出 mysql 程序，把原来的数据文件移回当前数据库子目录，用它替换掉你刚创建的空白数据文件。如果你在第(2)步还移动了索引文件，别忘了把它也移回到当前数据库子目录。

(5) 按照本节开头部分给出的步骤再次尝试修复数据表。

3. 让**myisamchk**程序运行得更快

myisamchk 程序需要花费较长的时间来运行，在修复一个大数据表或是在进行高级检查或修复的时候就更是如此。但我们可以通过让 myisamchk 程序在运行时使用更多内存来加快这个过程。myisamchk 程序有几个可以设置的变量，其中最重要的几个控制着它在运行时使用的缓冲区的大小（见下表）。

变量	含义
key_buffer_size	用来存放索引块的缓冲区的长度
read_buffer_size	用来完成读操作的缓冲区的长度
sort_buffer_size	用来完成排序操作的缓冲区的长度
write_buffer_size	用来完成写操作的缓冲区的长度

这几个变量的默认值可以通过使用--help 选项运行 myisamchk 程序的办法来查出。如果想把这几个变量设置为其他的值，需要在 myisamchk 命令行上使用--var_name=value 语法。比如说，如果有足够的内存，如下所示的命令将把 myisamchk 程序的排序缓冲区的长度设置为 16MB，把它的读缓冲区和写缓冲区的长度分别设置为 1MB：

```
% myisamchk --sort_buffer_size=16M --read_buffer_size=1M \
    --write_buffer_size=1M other-options tbl_name
```

在修复数据表时，--sort_buffer_size 值适用于--recover 选项，但不适用于--safe-recover 选项。--key_buffer_size 值适用于--safe-recover 修复选项和--extend-check 检查选项。

14.7　使用备份进行数据恢复

数据恢复工作需要两个信息来源：备份文件和二进制日志。备份文件既可以是用 mysqldump 程序生成的转储文件，也可以是采用某种二进制备份方法制作出来的副本。

备份文件能把数据表恢复到当初制作备份时的状态。二进制日志文件里记载着在制作备份以后执行过的修改数据表的语句。mysqlbinlog 程序能够把这些日志文件转换回文本形式的 SQL 语句，只需在 mysql 程序加以执行，就可以让那些发生在制作备份之后、出现问题之前的数据修改重现在各有关数据表里。

数据恢复工作的具体步骤取决于需要恢复的信息量有多少。事实上，因为针对整个数据库使用二进制日志要比只针对某个特定的数据表使用二进制日志更容易，所以恢复一个数据库通常要比只恢复一个数据表更容易。

接下来的讨论需要假设你已经制作了数据库备份并启用了二进制日志功能。如果不是这样，你就未免太危险了。在继续阅读本书之前，你应该立刻启用二进制日志功能并制作一份最新的备份。要知道，数据恢复工作离不开恢复数据所需要的信息，如果因为懒惰而没有保存那些信息，数据表一旦丢失就很可能再也找不回来了。启用二进制日志功能的具体办法见 12.5.4 节，制作数据库备份的具体办法见 14.3 节。

14.7.1　恢复整个数据库

恢复一个或多个数据库的基本步骤如下所示。

(1) 为数据库子目录制作备份。万一你在恢复过程中犯了错误，这些备份可以让你有机会从头再来。

(2) 使用最新的备份文件重新加载你打算恢复的数据库。

- 如果备份文件是用mysqldump程序生成的转储文件，按顺序使用各个备份文件作为mysql程序的输入加载之。

  如果需要恢复的数据库包括 mysql 数据库（其内容是各种权限数据表），并且你打算使用备份文件来恢复那些数据表的话，你将需要使用--skip-grant-tables 选项才能重新加载它们。否则，MySQL 服务器将抱怨“无法找到权限数据表”。还有个好主意是使用--kip-networking 选项让 MySQL 服务器在你正在进行恢复的过程中拒绝所有的远程连接。等你把数据表都恢复好以后，关停 MySQL 服务器再像往常那样重新启动它，就可以让它正常地使用权限数据表和监听网络接口了。
- 如果使用的是二进制备份（比如说，用mysqlhotcopy、tar或cp程序制作的备份），必须先关停MySQL服务器以防止它在你正在恢复某个数据表的同时访问该数据表，再把备份文件复制到它们原来的位置（通常都在MySQL数据目录下），然后重新启动MySQL服务器。

(3) 根据二进制日志重新执行在制作备份后对数据进行修改的 SQL 语句。具体步骤见 14.7.3 节。

14.7.2　恢复数据表

恢复一个数据表往往要比恢复一个数据库更困难。如果你有一个用由 mysqldump 程序生成的只包含那个数据表的转储文件，重新加载那个备份文件即可。如果你的转储文件包含来自多个数据表的数据而你只想恢复其中之一，你可以先通过编辑文件的办法把其他数据表里的数据全部删掉，然后再重新加载剩下来的数据。这一步还是比较容易的。

接下来的步骤就比较困难了：从二进制日志里把与那个数据表有关的语句不多不少地提取出来。mysqlbinlog 程序倒是有一个--database 选项可以让该程序只输出与某给定数据库有关的语句，但它没有针对单个数据表的相应选项。在遇到这种局面的时候，一个比较实用的策略是“先多恢复一些数据，再丢弃你不需要的东西”。与只从二进制日志里提取与给定数据表有关的语句相比，这个策略要容易实施得多。

(1) 先另行创建一个空白的数据库，然后把你打算恢复的那个数据表所在的数据库全部恢复到这个空白的数据库里。你可以用加载备份文件和重新执行二进制日志的办法来完成这次恢复，但有两个细节必须注意。

- 由mysqldump程序生成的转储文件可能包含一条初始数据库的USE语句。你需要修改或者干脆删除这条USE语句，再把转储文件用作mysql程序的输入。
- mysqlbinlog程序的输出可能包含一条或多条初始数据库的USE语句。先把mysqlbinlog程序的输出保存为一个临时文件，再通过编辑那些语句命名第二个数据库，然后再把那个临时文件用作mysql程序的输入。

(2) 在新创建的数据库里，用 mysqldump 程序转储你真正想要恢复的那个数据表。

(3) 删除原来的数据表，把转储文件加载到原来的数据库以重新创建那个数据表。如果你运行 mysqldump 程序时使用了--opt 或--add-drop-table 选项的话，转储文件里将会有一条 DROP TABLE 语句，该语句的作用是删除给定的数据表，然后重新创建。

对于 MyISAM 数据表，还有一个办法是用 mysqldump 程序把有关的数据表文件从第二个数据库子目录直接复制到原来的数据库子目录。需要特别注意的是，在进行复制操作时一定要保证那两个版

本都没被 MySQL 服务器使用着。

14.7.3 重新执行二进制日志文件里的语句

在使用备份文件恢复数据库或数据表之后，还应该把二进制日志文件里制作备份以后执行过的语句重新执行一遍。这样才能把数据表进一步恢复到最新的状态。

mysqlbinlog 程序可以把二进制日志文件转换回文本形式的 SQL 语句，让它们容易执行：把 mysqlbinlog 程序的输出用作 mysql 程序的输入。

根据此前使用备份都恢复了哪些东西，你可能需要执行二进制日志文件里的所有语句，也可能只需要执行与某个特定的数据库有关的语句，甚至可能需要选择只执行在一个特定的时间区间内执行过的语句。这些事情都可以交给 mysqlbinblog 程序去做。它可以处理多个二进制日志文件，可以限制它自己只输出与某给定数据库有关的语句或是在给定时间区间内执行过的语句。

在接下来的讨论内容里，我们将假设二进制日志文件的命名格式是 binlog.nnnnnn，文件扩展名 nnnnnn 由六位数字构成，它代表着日志文件的顺序编号。在你们自己的系统上，如果二进制日志文件的基本名不是 binlog，请在下面的示例命令里替换。此外，这里的讨论重点是如何使用 mysqlbinlog 程序来处理保存在本地主机里的二进制日志文件。mysqlbinlog 程序有能力读取远程二进制日志文件，但我们不准备在这里讨论它；如果你想了解这个工具程序的远程日志处理选项的细节，请参阅附录 F。

如果恢复数据库的备份是在创建所有二进制日志文件之前完成的，你还需要执行所有二进制日志文件的内容。具体做法是：切换到那些二进制日志文件所在的子目录，然后执行如下所示的命令：

```
% mysqlbinlog binlog.[0-9]* | mysql
```

如果需要先编辑日志文件再重新执行它们，可以先把它们转换为文本格式并把转换结果保存为一个临时文件，再编辑那个临时文件，然后再把编辑结果馈入 mysql 程序。下面是一个例子：

```
% mysqlbinlog binlog.[0-9]* > text_file
% vi text_file
% mysql < text_file
```

这个策略非常有用，尤其是如果你执行恢复和使用日志进行数据恢复的原因是因为某个用户不小心发出了 DROP DATABASE、DROP TABLE 或 DELETE 语句。在开始执行日志文件的内容之前，必须先把那些“肇事”语句从日志中剔除出去。

在上面的例子里，mysqlbinlog 命令行中的 binlog.[0-9]*模板将扩展为二进制日志文件的名单，各日志文件在名单里的先后顺序通常与 MySQL 服务器生成它们的顺序相同。

不要使用 mysqlbinlog 和 mysql 一个个地处理二进制日志文件。这是因为各日志文件之间可能存在着某种依赖关系，如果不把它们当做一个整体来处理，就有可能破坏那些依赖关系。比如说，后面的日志文件往往会用到前面的日志文件所创建的 TEMPORARY 数据表，如果分别处理每个日志文件，各日志创建的 TEMPORARY 数据表都会在相应的 mysql 执行完毕时被自动删除，在后面的日志中的语句里不再可用。

如果只需要把作用于某特定数据库的语句提取出来，可以借助于 mysqlbinlog 程序的 --database 选项：

```
% mysqlbinlog --database=db_name binlog.[0-9]* | mysql
```

mysqlbinlog 程序还有几个选项可以用来提取在给定时间区间执行过的语句（比如说，在制作了一个给定的备份之后才被写入日志文件的语句）。你可能需要先查看日志文件的内容才能决定应该使用哪些选项。下面是 mysqlbinlog 程序的一个输出样板（为了适应本书的页面宽度，我们对一些注释语句进行了删节）：

```
...
# at 1077
#071030 16:50:36 server id 1  end_log_pos 106   Query....
SET TIMESTAMP=1193781036;
INSERT INTO absence VALUES (3,'2008-09-03');
# at 1183
#071030 16:50:36 server id 1  end_log_pos 1210  Xid = 386
COMMIT;
# at 1210
#071030 16:50:36 server id 1  end_log_pos 106   Query....
SET TIMESTAMP=1193781036;
INSERT INTO absence VALUES (5,'2008-09-03');
# at 1316
#071030 16:50:36 server id 1  end_log_pos 1343  Xid = 387
COMMIT;
# at 1343
#071030 16:50:36 server id 1  end_log_pos 107   Query....
SET TIMESTAMP=1193781036;
INSERT INTO absence VALUES (10,'2008-09-06');
# at 1450
#071030 16:50:36 server id 1  end_log_pos 1477  Xid = 388
COMMIT;
...
```

如果想重新执行从 2007-10-30 16:50:36 那一刻开始被记载到二进制日志里的修改，可以使用 mysqlbinlog 程序的--start-datetime 选项来给出那个时间值，具体做法有以下两种：

```
% mysqlbinlog --start-datetime=20071030165036 binlog.[0-9]* | mysql
% mysqlbinlog --start-datetime="2007-10-30 16:50:36" binlog.[0-9]* | mysql
```

mysqlbinlog 程序还有一个相应的--stop-datetime 选项可以用来给出一个截止时间，它还有一个基于位置的选项可以用来给出日志里的 log_pos 值。详见附录 F 里对 mysqlbinlog 程序的讨论。

14.7.4　InnoDB存储引擎的自动恢复功能

在 MySQL 服务器或服务器主机崩溃以后，InnoDB 存储引擎将在 MySQL 服务器重新启动时尝试进行自动恢复。这种自动恢复一般都能成功，但在极少数情况下会失败。本节将讨论万一 InnoDB 存储引擎的自动恢复失败了应该怎么办。

如果 InnoDB 存储引擎在服务器重启过程中检测到一个无法恢复的问题，它的自动恢复过程将失败。在发生崩溃后，如果 InnoDB 存储引擎的自动恢复过程真的失败了，请把 innodb_force_recovery 系统变量设置为 1 到 6 之间的一个非零值并强行启动 MySQL 服务器。设置这个变量的办法很简单，只要在 MySQL 服务器的选项文件的[mysql]组里加上一行如下所示的代码即可：

```
[mysqld]
innodb_force_recovery=level
```

InnoDB 存储引擎默认使用的自动恢复策略比较保守，它会尽量选用一个比较小的 level 值。在

需要强行启动 MySQL 服务器的绝大多数场合，建议大家从 4 开始设置 `level` 值。在强行启动 MySQL 服务器之后，应该先用 `mysqldump` 程序转储 InnoDB 数据表以尽量减少信息丢失，然后删除 InnoDB 数据表，再用 `mysqldump` 输出文件恢复它们。按照这一过程重建出来的 InnoDB 数据表可以保证其内容数据的一致性，恢复效果往往能够令人满意。在完成上述恢复过程之后，别忘了从选项文件里去掉用来设置 `innodb_force_recovery` 变量的那行代码。

如果需要恢复所有的 InnoDB 数据表，就肯定要用到备份。而具体的恢复步骤要决于你当初制作的备份是什么类型的。

- 如果当初制作的是二进制备份，它应该包括共享表空间文件和专用表空间文件、InnoDB日志文件、每个InnoDB数据表的`.frm`文件以及用来定义InnoDB配置的选项文件。在确认MySQL服务器已被关停之后，把现存的InnoDB文件全部删掉并用备份副本替换它们。然后，查看当前的服务器选项文件并确保它列出的InnoDB配置与当初备份的选项文件保持一致。最后，重新启动MySQL服务器。
- 如果当初是用`mysqldump`程序来备份InnoDB数据表生成一个转储文件。你应该重新初始化共享表空间和InnoDB日志并把转储文件重新加载到InnoDB。

  (1) 关停 MySQL 服务器并把现存的 InnoDB 文件全部删掉，这包括：共享表空间文件和专用表空间文件（但不包括原始的硬盘分区）、InnoDB 日志文件以及每个 InnoDB 数据表的`.frm` 文件。

  (2) 按原先的设置重新配置共享表空间，然后重新启动服务器。InnoDB 存储引擎将重新创建它的共享表空间和日志文件。具体步骤见 12.7.3 节的第 1 小节。请注意，如果使用了硬盘分区，对表空间进行初始化将包含两个步骤。

  (3) 把当初制作的转储文件用作 `mysql` 程序的输入以重新加载它们。这将重新创建 InnoDB 数据表。

用备份恢复了 InnoDB 数据表之后，还要把二进制日志里加载的在制作备份后执行过的语句重新执行一遍，具体做法见 14.7.3 节。如果恢复 InnoDB 数据表是全面恢复一个或多个数据库的环节之一，事情就简单了，只要简单地把制作备份后执行过的所有修改执行一遍就可以完成任务。如果是只恢复 InnoDB 数据表，事情会稍微麻烦一些，因为你将需要使用只与那几个数据表有关的修改。

14.8　设置复制服务器

建立数据库复制的办法之一，是把某原始数据库复制到另一个 MySQL 服务器去。可是，如果这个原始数据库的内容变化了而你又想让它的复制品同样变化，就不得不重复操作。要想让复制品能够在原始数据库的内容发生变化时及时作出相应的修改，就要用到 MySQL 的实时复制机制。这一机制使我们能够随时拥有原始数据库的一份副本，并在原始数据库的内容发生变化时让它们自动而且及时地在副本上反映出来。

14.8.1　复制机制的工作原理

MySQL 数据库系统中的复制机制遵循以下原则。

- 在复制关系中，一个MySQL服务器将扮演“主人”角色（即所谓的“主MySQL服务器”，以下简称“主服务器”），另一个则扮演“仆从”角色（即所谓的“从MySQL服务器”，以下简称“从服务器”）；“仆从”将严格按照“主人”的一举一动行事。你必须给这两个MySQL服务器

分别分配一个独一无二的复制ID。

- 每个主服务器可以有多个从服务器。一个从服务器可以用作另一个从服务器的主服务器，也就是说，我们可以创建一个复制服务器链。把多个主服务器复制为同一个从服务器也是可以的，但设置步骤相当麻烦，本节将不对此做进一步讨论。
- 在复制关系形成之初，主服务器与从服务器必须完全同步——即该复制关系所涉及的各有关数据库在这两个MySQL服务器上必须有着完全一样的内容。此后，人们在主服务器上作出的修改动作将被传输并实现在从服务器上，但人们在主服务器上作出的修改动作并不直接作用在从服务器里的复制数据库上。
- 负责在主、从服务器间传输各种修改动作的媒介是主服务器的二进制日志，这个日志记载着需要传输给从服务器的各种修改动作。因此，主服务器必须启用二进制日志功能。存储在二进制日志里的修改被称为“事件”（event）。
- 每个从服务器都必须具备让它连接主服务器并请求修改的权限。当一个从服务器连接到它的主服务器时，它会告诉主服务器它在上次连接时已经进展到主服务器的二进制日志里的什么位置。这个进展是用复制坐标来表示的：一个二进制日志文件的名字和该文件里的一个位置。主服务器将从其二进制日志里的给定坐标位置开始把在那以后发生的事件传送给从服务器。当从服务器读取完主服务器的所有事件时，它将暂停并进入等待状态。
- 当主服务器上发生新的数据修改操作时，它将把它们记入它的二进制日志以便向从服务器传输。
- 主服务器对从服务器的处理和它对待普通客户程序差不多，而每一个连接到主服务器的从服务器都将占用`max_connections`系统变量所设置的系统最大连接个数。
- 在从服务器端，服务器使用两个线程来完成复制任务。I/O线程负责接收来自主服务器的待处理事件并把它们写入从服务器的中继日志（relay log）。SQL线程负责从中继日志读出事件并执行它们。中继日志相当于I/O线程向SQL线程传输数据修改信息的通信纽带。在处理完一个中继日志文件之后，从服务器将自动删除它。I/O线程和SQL线程的操作是互不影响的，它们可以分别启动和关停。两个线程在功能上的这种“脱节”有重要的好处。比如说，我们可以让I/O线程继续读取来自主服务器的事件，同时把SQL线程停下来制作数据库备份，这样既不会影响从服务器接收信息，又可以保证从服务器里的数据库在我们制作备份时不会发生变化。

复制支持功能是专家学者们仍在研发的一个领域，我们很难预知会有哪些与复制机制有关的新功能会在何时被添加到 MySQL 软件里。作为 MySQL 软件的使用者，我们必须考虑不同版本的 MySQL 服务器对复制机制的支持是否兼容。复制机制的兼容性以 MySQL 服务器的二进制日志的格式为基础，该格式目前有好几种版本。最早的版本是在 MySQL 3.23 里开发的，那是第一个支持复制机制的 MySQL 服务器，另外几种格式是在 MySQL 4.0、5.0 和 5.1 里开发的。

一般来说，我建议大家按以下原则行事。

- 在一个给定的MySQL版本系列（5.0、5.1等）内，要尽量选用最新的版本。相对而言，同一系列中的最新版本往往有着最丰富的功能、最少的局限性以及最少的程序漏洞。
- 尽量让主服务器和从服务器使用相同格式的二进制日志。比如说，如果主服务器是5.1版，从服务器就应该选用5.1版的而不是选用5.0版的，反之亦然。万一你的主、从服务器的版本不匹配，应该让主服务器的版本低于从服务器的版本，除此之外没有其他好办法。

复制机制中的主、从服务器不仅要在二进制日志的格式方面兼容，还需要在功能上兼容。比如说，

如果主服务器里有要求使用事务或外键的 InnoDB 数据表，从服务器就必须带有 InnoDB 存储引擎。

14.8.2 建立主从复制关系

在两个 MySQL 服务器之间建立主从复制关系的具体步骤如下。

(1) 确定自己想给这两个 MySQL 服务器分配什么样的 ID 值并把它们记录到 MySL 服务器在启动时会去读取的某个选项文件里。主从服务器的 ID 值应该是 1 到 $2^{32}-1$ 之间的一个正整数并且必须彼此不同。在启动主、从服务器的时候，你必须用 server_id 启动选项给出其 ID 值。在此基础上，一定要启用主服务器上的二进制日志功能——如果以前没有启用过它。要想在主、从服务器上启用二进制日志，最省心的办法是在相应的选项组里加上如下所示的代码：

```
[mysqld]
server-id=master_server_id
log-bin=binlog_name

[mysqld]
server-id=slave_server_id
```

重新启动两个服务器，让改动生效。

(2) 在主服务器上，创建一个账户供从服务器连接主服务器并请求修改信息：

```
CREATE USER 'slave_user'@'slave_host' IDENTIFIED BY 'slave_pass';
GRANT REPLICATION SLAVE ON *.* TO 'slave_user'@'slave_host';
```

请记住 slave_user 和 slave_pass 值，因为稍后还需要使用它们来告诉从服务器如何连接主服务器。如果这个账户的用途只用于复制，就用不着再向它授予任何其他的权限了。不过，为了对复制机制进行测试，你通常要在从服务器上用 mysql 程序以“手动方式”去连接主服务器，所以你往往还应该再给这个账户多授予几项权限以方便自己能够多做一些事情。（比如说，如果这个账户仅有 REPLICATION SLAVE 权限的话，你甚至无法在从服务器上使用 SHOW DATABASES 语句去查看主服务器上的数据库名称。）

(3) 连接到主服务器并通过执行 SHOW MASTER STATUS 语句确定当前的复制坐标：

```
mysql> FLUSH TABLES; SHOW MASTER STATUS;
+---------------+----------+--------------+------------------+
| File          | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+---------------+----------+--------------+------------------+
| binlog.000093 |     1707 |              |                  |
+---------------+----------+--------------+------------------+
```

请记住 File 和 Position 值，因为稍后还需要使用它们来告诉从服务器从哪个位置开始去读取主服务器的二进制日志里的事件。

重要注意事项 在主服务器上，在从确定其复制坐标到制作出将被传输到从服务器的初始复制的这段时间里，一定要保证主服务器上的数据库里的数据没有发生任何修改。

(4) 在从服务器上为将被复制的数据库建立一份完备的副本。把主服务器上的数据库复制到从服务器以完成主、从服务器之间最初的同步。一种办法是先在主服务器主机上制作一份备份，再把这个备份加载到从服务器上去；另一种办法是通过网络把各数据库从主服务器全部复制到从服务器。对数

据库备份技术和复制技术的讨论见本章的其他小节。

如果你还没有在主服务器上创建过任何数据库或数据表，可以省略这一步，因为它现在还没有任何需要复制的东西。

(5) 连接到从服务器并使用 CHANGE MASTER 语句来配置它，这包括把用来连接主服务器的参数和初始复制坐标告诉从服务器：

```
CHANGE MASTER TO
  MASTER_HOST = 'master_host',
  MASTER_USER = 'slave_user',
  MASTER_PASSWORD = 'slave_pass',
  MASTER_LOG_FILE = 'log_file_name',
  MASTER_LOG_POS = log_file_pos;
```

'master_host'是主服务器的主机名。'slave_user'和'slave_pass'值是刚才在主服务器创建的那个账户的用户名和口令，从服务器将使用这个账户来连接主服务器并请求主服务器修改信息。'log_file_name'和'log_file_pos'是刚才用 SHOW MASTER STATUS 语句查出来的值。

在 Unix 系统上，使用 localhost 作为主机名将使用一个套接字文件去连接主服务器，但复制机制不支持经套接字文件建立的连接。因此，如果主服务器和从服务器将运行在同一台主机上，就必须把这个主机名写成 127.0.0.1 而不是 localhost，这样才能确保从服务器将使用 TCP/IP 连接。

如果主服务器所监听的网络端口不是默认端口，可以在 CHANGE 语句里包括一个 MASTER_PORT 选项以明确地给出一个端口号。

(6) 让从服务器开始复制。

```
START SLAVE;
```

从服务器将连接主服务器并开始复制。你可以在从服务器上使用 SHOW SALVE STATUS 语句来查看它的工作状态。

从服务器把 CHANGE MASTER 语句所给出的参数保存在其数据目录中的一个名为 master.info 的文件里以记录初始复制状态，并随着镜象工作的进展而刷新那个文件。等以后需要改变复制参数的时候，只要连接到从服务器并通过 CHANGE MASTER 语句给出新的设置，从服务器就会根据新的设置自动刷新 master.info 文件。

保存在 master.info 文件里的信息包括用来连接主服务器的用户名和口令。这些信息应该是保密的，所以应该把这个文件设置为只允许从服务器上的 MySQL 管理员的登录账户才能访问。比如说按 13.1.2 节描述的，锁定数据目录内容。

以上步骤适用于你打算把主服务器上的所有数据库——包括容纳着各种权限表的 mysql 数据库在内——都复制到另一个 MySQL 服务器上的情况。如果不想让你的主、从服务器有完全相同的账户信息（比如说，你或许只想建立一个私用的复制从服务器，不想让在主服务器上有账户的人都能随意连接那个从服务器），可以把 mysql 数据库排除在复制机制外，这需要做两件事情。

(1) 在把数据库的初始数据从主服务器传输到从服务器的时候，不要把 mysql 数据库也包括在内。另一个办法是在传输之前先备份从服务器的 mysql 数据库，等传输工作完成后再恢复它。

(2) 在从服务器的选项文件里加上以下代码，让它不要执行来自主服务器的对 mysql 数据库的任何修改：

```
[mysqld]
replicate-ignore-db=mysql
```

如果想让从服务器忽略多个数据库，需要多次使用 replication_ignore_db 选项——每个数据库一次。

还有一个办法是在主服务器端（注意，不是从服务器端）使用--binlog-ignore-db 选项排除数据库。这个办法的优点是可以减少从主服务器传输到从服务器的信息量，缺点是主服务器上的二进制日志文件将不包含那些被排除在外的数据库的任何信息，而这些信息对主服务器在发生崩溃后的数据恢复工作往往至关重要。因此，在从服务器端把数据库排除在外的做法更值得选择。

在建立起复制关系并使之开始运转之后，还需要对主、从服务器进行监控和管理，这些工作可以用以下语句来完成。对这些语句的详细介绍见本书的附录 E，下面只是对它们的简单介绍。

- SHOW SLAVE STATUS语句。在从服务器上查看其复制机制是否在工作以及当前的复制坐标。复制坐标可以用来判断主服务器上的哪几个二进制日志文件已经不再会被用到了。
- PURGE MASTER语句。在主服务器上对二进制日志文件进行失效处理。在每一个从服务器上都通过SHOW SLAVE STATUS语句确定了哪些日志文件不会再用之后，你可以在主服务器上用这条语句把那些二进制日志文件删除掉。
- STOP SLAVE和START SLAVE语句。用来挂起和重新开始从服务器上的复制活动。比如说，当你制作备份时，可以用这些语句让从服务器暂时停止复制活动。（参见14.8.4节。）

如前所述，从服务器使用两个内部线程来管理复制活动。I/O 线程负责与主服务器进行通信，接收来自主服务器的信息，把接收到的数据修改命令写入从服务器的中继日志。SQL 线程负责从中继日志读出数据修改命令并执行。可以通过在 SLAVE STOP 或 SLAVE START 语句的末尾加上 IO_THREAD 或 SQL_THREAD 关键字的办法分别挂起或者重新开始这两个线程中的任何一个。比如说，STOP SLAVE SQL_THREAD 命令将使从服务器停止执行中继日志里的数据修改命令，但从服务器仍能继续接收来自主服务器的数据修改命令并把它们记载到中继日志里。

类似于二进制日志文件，从服务器将把中继日志生成为一系列按数字编号的文件，并且也会有一个类似于二进制日志索引文件的中继日志索引文件。默认的中继日志文件和索引文件分别是数据目录里的 HOSTNAME-relay-bin.nnnnnn 和 HOSTNAME-relay-bin.index。这些默认使用的文件名可以通过从服务器的启动选项--relay-log 和--relay-log_index 加以改变。

14.8.3　二进制日志的格式

在 MySQL 5.1 版之前，服务器将把数据修改事件以 SQL 语句的形式写入二进制日志。这称为基于语句的二进制日志功能，相应的复制被称为基于语句的复制。从 MySQL 5.1.5 版开始，增加了一种新的格式，服务器使用这种格式把各个数据行的修改情况写入日志。这被称为基于数据行的日志功能，相应的复制被称为基于数据行的复制。从 MySQL 5.1.8 版开始允许使用混合格式的日志，服务器会在它认为最适当的时候切换使用基于语句的日志功能或者基于数据行的日志功能。

一般来说，基于语句的日志功能所生成的日志文件比较短小，内容也比较容易理解。基于数据行的日志功能在将要执行哪些修改方面可以提供更细致的控制。更细致的控制对复制来说是件好事，这是因为有些语句可能不够明确，在主服务器和从服务器上可能会有不同的执行效果。

从 MySQL 5.1 版开始，日志格式可以通过在启动时使用--binlog-format 选项或是（从 MySQL 5.1.8 版开始）通过在运行时设置 binlog_format 系统变量的办法来加以选择。从 MySQL 5.1.2 版开始，默认的日志格式是 MIXED（混合）。其他的可取值是 STATEMENT（语句）或 ROW（数据行），它们将强行选择一种给定的日志格式。

14.8.4　使用复制机制制作备份

复制机制中的从服务器可以在 MySQL 管理员面临下面这样的两难选择时帮上大忙。

- 一方面，MySQL管理员的一个重要职责是尽最大努力保证用户随时都能使用MySQL服务器，包括允许用户对数据库里的数据进行修改。
- 另一方面，MySQL管理员的另一个重要职责是制作备份，而这项工作最好是在任何人都不能对数据库里的数据进行修改时进行。此外，为了让数据恢复工作取得最好的效果，让备份文件和日志文件所覆盖的时间区间保持一致也非常关键，而这往往需要彻底关停MySQL服务器或是同时锁定所有的数据表。

让用户随时都能使用 MySQL 服务器和在备份时最大限度地限制客户对数据库的访问是两个相互冲突的目标，复制机制中的从服务器提供了一个解决方案：把备份工作放在从服务器上进行，不再使用主服务器来制作备份。在开始制作备份之前，关停从服务器或是暂停从服务器上的复制活动。等到把备份制作好以后，重新启动从服务器或是重新开始从服务器上的复制活动，从服务器将补上在备份期间发生在主服务器上的数据修改。这样一来，MySQL 管理员在备份时就用不着关停主服务器或者限制客户对主服务器的访问了。

下面是在从服务器上制作数据库备份的几种策略。

- 如果打算为从服务器上的所有数据制作一个二进制备份，可以关停从服务器，按照14.4.2节中的第1小节给出的步骤制作备份，重新启动从服务器。
- 如果打算采用一种无需关停从服务器的备份方法（比如使用`mysqldump`程序），可以在备份期间暂时停止从服务器的SQL线程，等备份制作好以后再重新启动SQL线程：用`STOP SLAVE SQL_THREAD`语句暂停从服务器上的复制活动并刷新它的日志，然后开始制作备份，最后用`START SLAVE`语句重新开始复制活动。这样一来，在制作备份的过程中，从服务器将不会对数据库进行任何修改。I/O线程可以保持运行，它将把它从主服务器那里接收到的复制事件继续写入中继日志。在备份工作结束之后，重新启动SQL线程，它将补上备份期间主服务器所做的任何修改。可以采用这个策略的前提是没有客户在从服务器上执行数据更改操作。请注意，如果打算使用一种二进制备份方法来直接复制数据库文件，千万不要使用这个策略。这是因为，虽然SQL线程被关停了，但可能会有一些缓存在内存里的信息没来得及转储到硬盘。
- 有些备份方法甚至不需要你暂停复制活动。比如说，如果只想备份一个只包含 MyISAM 数据表的数据库的话，可以用 `mysqlhotcopy` 或者 `mysqldump` 程序的适当选项同时锁住所有的数据表。在这些情况下，从服务器将保持运行，但它不会在你制作备份的过程中对已被锁定的数据表进行任何修改。当你完成备份工作并释放刚才锁定的数据表时，从服务器将自动地重新开始复制活动。

Part 4

第四部分

附　　录

本部分内容

附录 A

获得并安装有关软件

本书里的示例都取材于一个名为 sampdb 的示例数据库。本附录将介绍如何获得 sampdb 发行版本。为了使用这个发行版本，我们还得让 MySQL 系统运转起来。因此，本附录还将介绍如何获得和安装 MySQL 及其相关软件，比如 Perl DBI 和 CGI.pm 模块、PHP、Apache 等。本附录内容覆盖了 Unix 和 Windows 这两大类系统。

本附录的目的是把这里讨论的每个软件包的安装办法集中起来以方便大家查阅和参考，而不是想取代各种软件中自带的安装指南。事实上，我鼓励大家去阅读那些指南。虽说本附录提供的信息应该能够满足大多数场合的需要，但软件自带的安装指南对大家在标准安装过程中遇到的问题提供了更有针对性的解决方案。就拿 MySQL 手册来说吧，它有一章全面地介绍了安装过程，针对各种特定于操作系统的问题提供了解决方案。

A.1 如何获得示例数据库 sampdb 的发行版本

sampdb 发行版本包含了建立和访问 sampdb 数据库所需要的各种文件。可以在下面这网址上找到并下载它：

```
http://www.kitebird.com/ mysql-book/
```

sampdb 发行版本有 tar 压缩文件和 ZIP 压缩文件两种格式。tar 格式的发行版本可以用下面两条命令之一来解压缩（如果你的 tar 命令不支持 z 选项，请使用第二条命令）：

```
% tar zxf sampdb.tar.gz
% gunzip < sampdb.tar.gz | tar xf -
```

ZIP 格式的发行版本可以用 WinZip、pkunzip 或 unzip 等软件工具来解压缩。

在对 sampdb 发行版本进行解压缩的时候，它将创建一个名为 sampdb 的目录并在其中生成以下一些文件和子目录。

- 一个README.txt文件，包含使用sampdb发行版本的基本方法。这是第一个你应该阅读的文件。它的各个下级子目录里还会有一些内容更为具体的README.txt文件。
- 一些用来建立和加载sampdb数据库的文件。第1章用到了它。
- 一个capi子目录，其内容是第7章中的各种C语言示例程序。
- 一个perlapi子目录，其内容是第8章中的各种Perl DBI脚本程序。
- 一个phpapi子目录，其内容是第9章中的各种PHP脚本程序。
- 一个ssl子目录，其内容是用于在MySQL客户程序和服务器之间建立SSL连接的证书和键文件。

sampdb 子目录还有其他几个下级子目录，里面是一些在本书其他章节提到的有关文件。查看 README.txt 文件可了解更多信息。

A.2 如何获得 MySQL 及相关软件

为了学习本书，MySQL 软件是必不可少的；如果机器上还没有 MySQL，那你肯定得安装它。至于那些第三方软件，只要把你打算使用的那些安装上就行了。

- 如果想编写一些Perl脚本来访问MySQL数据库，就必须安装DBI和DBD::mysql模块。如果你还打算编写基于Web的DBI脚本，可能还得安装CGI.pm模块；当然还得有一个Web服务器才行。本书使用的Web服务器是Apache，但这并不表明其他Web服务器不能使用，你们大家可以随意选用。
- 如果想编写本书描述的PHP脚本，就必须安装PHP和PHP Data Object(PDO)数据库访问扩展。PHP主要用于编写基于Web的脚本，所以你肯定还得需要一个Web服务器。本书选用Apache服务器来配合PHP。

前面提到的这些软件大都有预编译好的二进制版本。如果你使用的是一个 Linux 系统，就有各种各样的 RPM 文件可供选用。如果你喜欢从源代码开始编译软件，或者找不到适用于你系统平台的二进制发行版本，那你还得准备一个 C 编译器（MySQL 软件要用一个 C++编译器来编译）。

如果在某个 ISP（Internet Service Provider，因特网服务提供商）处有账户，而这个 ISP 又提供有 MySQL 服务，那它很可能已经把以上这些软件都安装齐全了。如果是这种情况，你就可以直接去使用它们，而不必再研读本附录后面的内容了。如果不是这种情况，你就需要安装下面软件包的发行版本了。下面这些站点有几个镜像站点，它们提供的软件是相同的，但因为地理距离近，有些站点的下载时间要短得多。

软件包	网站
MySQL	http://dev.mysql.com/
Perl模块	http://cpan.perl.org/
PHP	http://www.php.net/
Apache	http://httpd.apache.org/

应该根据自己的具体情况来挑选软件包的具体版本和发行格式，这方面的考虑因素有以下几点。

- 如果想得到最大限度的稳定性，就应该保守地选择用软件包最新的稳定版本。这类版本不像开发版本那样有很多的试验性代码，它们既能提供最新的功能，又最大限度地修补了软件中的漏洞。
- 如果喜欢为新事物冒一点险，或者你需要的功能只有最新的开发版本才能提供，就应该选择最新的开发版本。
- MySQL的预生成二进制发行版本都是用优化选项建立的，其效果往往要比用源代码版本里的配置脚本里配置出来的效果更好。MySQL开发组推荐人们尽量使用来源于www.mysql.com站点的二进制发行版本。他们在构建一些发行版本时使用了一些商业化的优化编译器，从而使MySQL能够以更快的速度运行。因此，二进制发行版中的程序可能比你自己编译的那些运行得更快。此外，开发人员有丰富的经验来避免或处理阻碍MySQL正常工作的编译器或系统库bug。

这些软件包的官方站点会告诉你哪些是最新的稳定版本，哪些是当前的开发版本。这些站点为每

个版本提供的功能改进清单可以帮助你挑选到最合适的版本。

如果你决定选用一个二进制发行版本，解压缩工作就将相当于软件的安装工作，因为文件将被解压缩到预定的子目录里。在 Unix 上，你可能需要以 root 用户身份来进行解压缩操作，因为有些文件需要安装到你系统上的受保护子目录里去。

源代码发行版本的格式通常是压缩的 tar 文件或 zip 文件。你应该先把它们解压缩到一个临时子目录里，在那里完成编译工作后，再把软件安装到最终的安装地点去。在 Unix 上，你可能需要以 root 用户身份来安装，但配置和编译阶段的工作一般都用不着以 root 用户身份来进行。

如果打算在 Unix 系统上从源代码开始安装，本附录介绍的软件包中有几个可以用 configure 工具来配置，这个工具可以大大简化很多种系统平台上的软件安装和构建工作。如果构建失败，你就需要使用另外一些选项重新运行 configure 工具。但在此之前，需要先把 configure 工具在你上次使用它时保存起来的选项和有关信息清除掉。可以用下面这条命令来清除配置信息：

```
% make distclean
```

也可以使用下面这两条命令：

```
% rm config.cache
% make clean
```

订阅邮件列表以寻求帮助

在准备安装一个软件包时，先订阅一份与该软件有关的邮件列表是一个很不错的主意。一旦遇到麻烦，你就可以迅速提出问题并得到别人的帮助。如果你准备安装的是一个开发版本，那就更应该订阅有关软件的邮件列表了，它可以让你随时掌握最新的 bug 报告和修补措施。即使不想加入一个讨论组，也至少应该订阅它的通告列表，以便在第一时间获得新版本的发布信息。邮件列表的订阅和使用办法可以在说明书上查到，软件包的 Web 站点通常也会提供这方面的信息。

A.3　挑选一个 MySQL 版本

MySQL 软件有好几种发行版本系列。一般来说，应该选择打算使用的那个系列里编号最高的版本。就现时期而言，5.0 系列收录着稳定的版本，5.1 和更高的系列收录着仍在开发中的版本。本书内容围绕 MySQL 5.0 系列版本展开，但也涉及 5.1 和早期 6.0 系列版本里的功能。

如果没有特殊的要求，建议使用稳定的版本，避免使用开发中的版本，所以我推荐的是绝大多数读者应该选用的 MySQL 5.0。如果你想体验一下诸如事件调度器之类的新功能，那就选用 5.1 或更高的版本好了。（在我写作本书的时候，5.1 系列的稳定版本即将推出，到那时就可以在 5.0 或 5.1 系列当中作出选择而不必使用一个研发版本了。）

A.4　在 Unix 系统上安装 MySQL

Unix 系统上 MySQL 发行版本的格式有二进制码（预生成）源代码等几种。二进制码比较容易安装，但你必须接受别人为该发行版本预先安排好的子目录布局和默认配置。源代码格式的发行版本不太容易安装，因为你必须亲自编译有关软件，但你对配置参数有着更多的控制权。比如说，既可以只编译发行版本中你想要的存储引擎和字符集，也可以随心所欲地把软件安装到任何地方。

在接下来的内容里，我将讨论如何在 Unix 系统上安装以下类型的 MySQL 发行版本：

- tar 文件形式的二进制发行版本；

- ❑ 针对 Linux 操作系统的 RPM 软件包；
- ❑ tar 文件形式的源代码发行版本。

请注意，MySQL 软件还有其他的安装方法，比如使用 Mac OS X 系统上的 DMG 磁盘映像、FreeBSD 网络端口、Gentoo Linux 系统的 ebuild 功能或者是 Debian Linux 系统上的 `apt-get` 命令等。如果你正在使用的 Unix 系统或类似于 Unix 的系统有它自己的软件包管理机制，也可以使用它们来安装 MySQL 软件而不必拘泥于这里给出的步骤。（无论何种情况，完成初始安装工作之后都应该认真阅读 A.4.3 节里的内容。）

MySQL 发行版本通常由以下一个或者多个组件构成：

- ❑ `mysqld`服务器；
- ❑ 客户程序（`mysql`、`mysqladmin`等）；
- ❑ 客户程序编程支持（C语言开发库和各种头文件）；
- ❑ 语言支持；
- ❑ 文档；
- ❑ 基准校验数据库。

源代码和二进制码格式的发行版本通常都包含有上述全部组件。RPM 文件往往只包含上述组件中的一部分，因而需要安装多个 RPM 文件才能得到你想要的全部东西。

如果你只是需要连接到运行在另一台机器上的 MySQL 服务器去，那就不需要安装一个服务器。但你几乎总是需要安装上客户端软件，这样就可以连接到你使用的服务器。同样，如果要使用包含 C 客户端库的 API 编程，那么必须安装那个库。比如说，如果你想编写基于 MySQL 的 Perl 脚本，就肯定要用到 DBI，而 DBI 又要求你的机器上必须安装有 MySQL 的 C 语言客户程序开发库。

在 Unix 系统上安装 MySQL 软件的基本步骤如下所示。

(1) 如果打算安装一个 MySQL 服务器，就应该先创建一个系统登录账户，好让 MySQL 服务器能够以这个账户下的用户和用户组身份运行。（只有首次安装需要进行这项工作。如果是升级安装，这一步骤可以省略。）在 Linux 上，若需要，安装服务器 RPM 会自动创建登录账户。

(2) 获得并解压缩你想安装的 MySQL 发行版本。如果你打算从源代码开始安装，就需要对下载到的软件包进行编译和安装。

(3) 运行 `mysql-install-db` 脚本程序，对数据目录和各种权限数据表进行初始化。（只有首次安装需要进行这项工作。如果是升级安装，这一步骤可以省略。）有些发行版本会在你安装它们时运行 `mysql_install_db`。这包括 Linux 上的服务器 RPM 包和 Mac OS X 上的 DMG 包。

(4) 启动服务器。

(5) 阅读本书第 12 章，熟悉 MySQL 数据库系统的日常管理操作。具体而言，需要阅读有关如何启动和关闭服务器的那几节，以及如何用一个非超级用户账户来运行服务器的那几节。

A.4.1 为MySQL用户创建一个系统登录账户

只有首次安装或者要运行一个 MySQL 服务器时才必须创建登录账户。如果是升级安装或者是只安装 MySQL 的客户端软件，这一步骤可以省略。

MySQL 服务器允许你以 Unix 系统上任何一种用户身份来运行，但出于安全和管理方面的考虑，你不应该以根用户 `root` 身份来运行。我建议大家另外创建一个专用的系统登录账户来管理和运行 MySQL 服务器，这样就可以用那个账户登录并拥有对数据目录进行维护、管理和调试等操作所需要

的全部权限。有关使用 root 之外的其他账户的好处，请参阅 12.2.1 节中的第 1 小节。

用户账户的创建过程随系统的不同而有所变化，具体细节请参阅本地文档。(如果使用的是 RPM 软件包，MySQL 服务器部分的 RPM 安装过程会自动地替你创建一个用户名是“mysql”的登录账户。)

本书使用“mysql”作为 MySQL 系统管理账户的 Unix 用户名和 Unix 用户组名。如果安装 MySQL 软件的目的是仅供自己使用，你完全可以从用自己的账户来运行它，但这需要你在这本书里看到以“mysql”为用户名或用户组名的内容时把它替换为你自己的登录名和用户组名。

在开始创建一个用来运行 MySQL 的账户之前，应该先检查你的系统里是否已经有了一个这样的账户。有许多 Unix 系统在它们的标准账户清单里已经包括了一个“mysql”用户名和用户组名。比如说，新近推出的 Mac OS X 版本都包括了一个“mysql”账户（这就满足了在 Mac OS X 系统上使用 DMG 软件包安装 MySQL 软件时需要有一个已经存在的“mysql”账户的要求)。

A.4.2　获得并在Unix系统上安装MySQL发行版本

下面讨论如何安装不同发行版本的 MySQL。我将用 version 来代表你准备安装的 MySQL 发行版本的版本号，用 platform 来代表你打算安装 MySQL 软件的系统平台的名称。cpu 代表与发行版本对应的处理器类型。这些值都用在发行版本里的文件名中，这样就可以很方便地区分各个版本。这里所说的“版本号”指的是诸如“5.0.56”5.1.24-rc 或“6.0.5-alpha”之类的东西。“系统平台”、“CPU 名称”则指的是诸如“solaris10-sparc”的在 SPARC 硬件上运行 Solaris10 的系统，或“linux- i686”这样在 Intel 硬件上运行 Linux 的系统。

1. 安装tar文件形式的二进制发行版本

tar 文件形式的二进制发行版本的文件名是 mysql-version-platform-cpu.tar.gz 这样的。请根据想要的版本和系统平台获得相应的发行版本文件，并把它放到打算安装 MySQL 软件的子目录里(如/usr/local)。

```
% tar zxf mysql-version-platform-cpu.tar.gz
% gunzip < mysql-version-platform-cpu.tar.gz | tar xf -
```

解压发行版本文件会创建一个名为 mysql-version-platform-cpu 的目录，其中包含版本信息。为方便引用这个目录，创建一个名为 mysql 的符号链接。

```
% ln -s mysql-version-platform-cpu mysql
```

创建好链接之后，可以按/usr/local/mysql的格式引用安装目录。(假设将MySQL安装在了/usr/local 下面。现在可以进入 A.4.3 节。

2. 安装RPM发行版本

在基于 RPM 的 Linux 系统上，安装 MySQL 软件的工作可以使用 RPM 软件包完成。下面列出了一些比较常用的 RPM 软件包。在这些软件包的名字里，version 代表 MySQL 软件的版本号，platform 代表目标系统平台的类型（或者是“glibc23”，其含义是该软件包普遍适用于所有支持 glibc 2.3 库的 Linux 发行版本)，cpu 代表该发行版本所适用的处理器类型。

- MySQL-server-version-platform-cpu.rpm。MySQL服务器程序。
- MySQL-client-version-platform-cpu.rpm。MySQL 客户程序。如果从 RPM 包安装了 MySQL，就应安装客户软件包。
- MySQL-embedded-version-platform-cpu.rpm。嵌入式MySQL服务器libmysqld（MySQL 5.1 及更高版本）中含有。

- MySQL-devel-version-platform-cpu.rpm。开发支持包（客户程序开发库和各种头文件），用来编写客户程序。如果你打算编写一些能够访问MySQL数据库的C程序或Perl DBI脚本，就必须安装这个RPM文件。其他语言的MySQL API可能也取决于这些客户端库。
- MySQL-shared-version-platform-cpu.rpm。共享式客户端库。
- MySQL-bench-version-platform-cpu.rpm。基准校验和测试程序，要求安装有Perl DBI支持。（请参阅A.4.4节。）
- MySQL-version.src.rpm。服务器、客户以及各种基准校验和测验程序的源代码。

RPM 文件对其中各组件的安装位置已经做好了安排，所以你可以在任何一个子目录里来安装它们。给定一个 RPM 文件 rpm-file，你可以用下面这条命令来查知有关组件将被安装到哪些地方：

```
% rpm -qpl rpm-file
```

下面这条命令将安装 RPM 文件（你可能需要以 root 用户身份来做这件事）：

```
# rpm -i rpm-file
```

因为 MySQL 的各种组件被划分到了不同的 RPM 文件里，所以你通常需要安装一个以上的 RPM 文件才能得到你想要的全部东西。对于通常的安装，需要安装服务器和客户 RPM。如果只安装服务器 RPM，将不能用它完成很多事，因为它不包含客户程序。

对于服务器支持，使用这条命令：

```
# rpm -i MySQL-version-platform-cpu.rpm
```

下面这条命令将安装 MySQL 的客户程序：

```
# rpm -i MySQL-client-version-platform-cpu.rpm
```

如果需要利用 MySQL 客户程序开发支持包来编写一些程序，就还得安装相应的开发 RPM 文件：

```
# rpm -i MySQL-devel-version-platform-cpu.rpm
```

如果你打算从源代码 RPM 包开始安装 MySQL，下面这条命令应该能让你达到目的：

```
# rpmbuild --recompile --clean MySQL-version.src.rpm
```

3. 如何安装源代码格式的发行版本

源代码格式的 MySQL 发行版本都有一个 mysql-version.tar.gz 形式的文件名，其中的 version 是 MySQL 软件的版本号。（和二进制发行版不同，源代码发行版适用于所有系统，所以文件名中没有 platform 或 cpu 值。）提取你想放置解压缩后的发行版的那个子目录。然后，用下面两条命令之一来解压缩这个发行文件（如果你的 tar 命令不支持 z 选项，请使用第二条命令）：

```
% tar zxf mysql-version.tar.gz
% gunzip < mysql-version.tar.gz | tar xf -
```

发行文件的解压缩操作将自动创建一个包含发行版内容的名为 mysql-version 的子目录。用下面这条命令把当前路径切换到这个子目录：

```
% cd mysql-version
```

既然使用的是源代码格式的 MySQL 发行版本，就必须先对它进行配置和编译之后才能安装 MySQL 软件。如果上面这几个步骤遇到了麻烦，请阅读《MySQL 参考手册》中有关安装的章节，特别是与你的计算机型号相对应的有关章节。

现在，用 configure 命令来配置发行版：

```
% ./configure
```

你可能需要在 configure 命令里安排一些配置选项。configure 命令的配置选项可以用下面这条命令查出来：

```
% ./configure --help
```

下面是 configure 命令一些比较常用的配置选项。

- --with-innodb、--without-innodb。支持或者不支持InnoDB存储引擎的使用。不支持InnoDB将使服务器更小，使用的内存更少。但如果要使用InnoDB数据表，就应包含这个存储引擎。
- --without-server。只建立客户端部分（客户程序或客户端库）。如果你只需要访问运行在另一台机器上的MySQL服务器，就应该加上这个选项。
- --with-embedded-server。准备安装嵌入式服务器库libmysqld。
- --with-yassl、--with-openssl、--with-ssl——配置发行版本时包含SSL支持。如果想在客户和服务器之间使用加密SSL连接，这个选项是必需的。在MySQL 5.1.11之前，使用--with-yassl或--with-openssal来选择ya SSL或OpenSSL。从MySQL 5.1.11起，使用--with-ssl来选择yaSSL，或使用--with-ssl=*path_name*来选择OpenSSL。路径名称表明OpenSSL头文件和库所在的位置。
- --prefix=dir_name。在默认的情况下，MySQL软件的根安装目录将是/usr/local，它的数据目录、客户程序、服务器、客户端库、头文件、用户手册页、语言支持文件等组件将分别被安装到该目录的var、libexec、bin、lib、include、man、share等下级子目录里。如果想为MySQL软件另行指定一个根安装目录，就需要使用--prefix选项。dir_name应该是一个完整路径名。比如说，如果你给出了“--prefix=/usr/local/mysql”选项，MySQL软件的各种组件就都将被安装到/usr/local/mysql子目录里去。
- --localstatedir=dir_name。这个选项用来改变数据目录的存储地点。如果你不想把MySQL的数据目录放在它的根安装目录里，就可以利用这个选项来改变它。dir_name应该是一个完整路径名。

在运行完 configure 命令后，下面两条命令将分别完成编译和安装工作：

```
% make
% make install
```

如果没有在 configure 命令里用--prefix 选项为 MySQL 指定一个你拥有其写权限的子目录作为它的根安装目录，你就可能需要以 root 用户身份来执行安装命令。

现在进入到 A.4.3 节。

A.4.3　后安装步骤

如果是首次安装，在安装好 MySQL 软件后还有几个应该完成的步骤。

(1) 设置 PATH 环境变量，把 MySQL 客户程序所在的子目录包括到其中。如果只安装了客户程序，并且无需运行服务器，那么只要完成这个步骤就行了，后面的步骤可以跳过。

(2) 对 MySQL 数据目录和权限数据表进行初始化。

(3) 启动 MySQL 服务器。

(4) 创建“地理时区”数据表，以便用户可以使用地理时区的名字。

(5) 创建服务器端“帮助”数据表。

如果是对现有的 MySQL 系统升级，上述步骤当中也许你已经完成了几个，如设置 PATH 环境变量和对 MySQL 数据目录进行初始化等。但你仍有可能需要做好以下几项工作。

- 升级权限数据表，以确保它们都有最新的结构。
- 创建“地理时区”数据表，因为 MySQL 4.1.3 版之前的老系统不支持多地理时区。这些数据表将支持使用地理时区的名字来设置时区。
- 创建服务器端“帮助”数据表，让用户参阅最新的帮助文本。

接下来的几个小节将对如何完成这些操作进行描述，请根据具体情况阅读和实践。

1. 设置PATH环境变量

如果想从命令行执行 MySQL 客户程序时不必输入完整路径名，就需要设置 PATH 环境变量把位于 MySQL 安装目录下的 bin 子目录包括进来。你的 shell（命令处理器）需要使用这个变量来确定应该到哪里寻找那些命令。PATH 变量通常在 shell 的某个启动文件里设置，如 tcsh 的.tcshrc 文件、bash 的 v.bashrc 或 v.bash_profile 文件等。比如说，如果使用的是 tcsh，在.tcshrc 文件里就应该有一行下面这样的代码：

```
setenv PATH /bin:/usr/bin:/usr/local/bin
```

如果把 MySQL 客户程序统一安装在/usr/local/mysql/bin 子目录里，像下面这样改变 PATH 变量的值即可：

```
setenv PATH /usr/local/mysql/bin:/bin:/usr/bin:/usr/local/bin
```

如果使用的是 bash，在它的某个或某几个启动文件里就应该有一行下面这样的代码：

```
PATH = /bin:/usr/bin:/usr/local/bin
```

把这项设置修改成下面这样：

```
PATH = /usr/local/mysql/bin:/bin:/usr/bin:/usr/local/bin
```

在完成对 shell 的启动文件的修改之后，新设置将在此后的每一次登录时生效。

2. 对数据目录和权限数据表进行初始化

刚安装好的 MySQL 软件还不能立刻投入使用，你还得先对 mysql 数据库进行初始化才行。在 mysql 数据库中的各种权限数据表里，存放着你 MySQL 服务器的各种访问权限控制信息。不过，只有首次安装或者在打算运行服务器时才必须进行这项工作。如果是升级安装或者是只安装 MySQL 的客户端软件，这一步骤可以省略。

在下面的讨论内容里，我将用 DATADIR 来代表你数据目录的路径名，通常位于 MySQL 安装基目录下，名为 data 或 var。一般说来，本小节里的命令都需要以 root 用户身份才能执行。但如果你是通过系统分配给 MySQL 的登录账户（如 mysql 用户）进入系统的或者把 MySQL 安装在了你自己的账户下（即你安装的 MySQL 只供你一个人使用），那你即便不是 root 用户也能执行这些命令。如果是这种情况，你将用不着执行下面的 chown 和 chgrp 命令。

要想对数据目录、mysql 数据库和默认的权限数据表进行初始化，你只需把当前路径切换到 MySQL 的根安装目录再执行 mysql_install-db 脚本就行了。（如果你当初是使用 RPM 文件或 Mac OS X 包来安装 MySQL 软件的，就不需要这一步骤，因为安装过程已经自动执行过 mysql_install-db 脚本了。）

比如说，如果你把 MySQL 软件安装到了/usr/local/mysql 目录里，就需要使用下面两条命令：

```
# cd /usr/local/mysql
# bin/mysql_install_db --user=mysql
```

如果 mysql_install-db 脚本执行失败，请参考《MySQL 参考手册》有关 MySQL 安装过程的章节，查找问题原因和解决方案。但有一点需要大家特别注意：如果 mysql_install-db 脚本没有成功地执行到结束，它创建出来的各种权限数据表就很可能是不完整的。如果 mysql_install-db 脚本发现这些权限数据表已经存在，就不会再去重新创建它们，因此，你应该删掉它们。下面这条命令可以把整个 mysql 数据库删除掉：

```
# rm -rf DATADIR/mysql
```

在成功地运行完 mysql_install-db 脚本之后，对数据目录里全体文件的属主信息（用户和用户组）和访问权限信息做出相应的修改。假设用户和用户组名字都是 mysql，使用下面的命令：

```
# chown -R mysql DATADIR
# chgrp -R mysql DATADIR
# chmod -R go-rwx DATADIR
```

先用 chown 和 chgrp 命令把文件的属主修改为 MySQL 登录用户和用户组，再用 chmod 命令把文件的访问模式设置为只允许那个用户访问，而拒绝其他任何用户访问。

3. 启动MySQL服务器

只有必须运行 MySQL 服务器的场合才需要进行这一步骤。如果你只安装了 MySQL 软件的客户程序部分，可以跳过这一小节。本小节里的命令都需要在 MySQL 的安装目录里执行（就像上一小节里的命令一样）。一般来说，本小节里的命令都需要以 root 用户身份才能执行。但如果你是通过 MySQL 登录账户（如 mysql 用户）进入系统的或者把 MySQL 安装在了你自己的账户下，那你即便不是 root 用户也能执行这些命令，如果是这种情况，请省略以下命令中的“--user”选项。

下面的第一条命令把当前路径切换到 MySQL 的根安装目录（以/usr/local/mysql 为例），第二条命令启动了 MySQL 服务器：

```
# cd /usr/local/mysql
# bin/mysqld_safe --user=mysql &
```

--user 选项的作用是让服务器以 mysql 用户来运行。

如果是第一次在这台机器上安装 MySQL，可能还得做以下几件事。

- ❑ MySQL软件的默认安装允许MySQL的root用户不使用口令来进行连接。为安全起见，你最好趁现在给它设置一个口令。
- ❑ 你可以把MySQL安排为整个系统的开、关机过程的一个组成部分，让它在系统开机时自动开始运行，在系统关机时自动结束运行。
- ❑ 你可以把--user选项放到某个选项文件里，这样，你就不必在每次启动MySQL的时候都把它敲入一遍。
- ❑ 有选择地启用各种日志功能。这对服务器的监视工作、复制和数据恢复工作都有好处。
- ❑ 你可以启用或禁用存储引擎，或为它们调整参数。

以上几种操作的具体步骤可以在第 12 章中查到。

4. 安装或升级其他的系统级数据表

如果是对一个现有 MySQL 系统升级，权限数据表的结构有可能已经发生了变化。如果需要升级

那些数据表，使用最新的结构，请按照 12.4 节的操作步骤。

如果需要对“地理时区”数据表进行设置，以便用户们使用时区的名字来调整时区设置，请按照 12.9.1 节的操作步骤。

mysql 命令行客户程序可以通过 help 命令访问服务器端的帮助信息，但 mysql 数据库里的“帮助”数据表必须已设置到位。绝大多数安装方法会在首次安装时自动完成这个设置，但万一你选用的方法没替你做好这件事，就需要手动加载“帮助”数据表了。为此，在确认 MySQL 服务器已经运行之后，找到 fill_help_tables.sql 文件。这个文件的内容是用来创建和加载“帮助”数据表的 SQL 语句，它通常位于/usr/share/mysql 子目录、MySQL 安装目录下的 share 子目录，或者某个源代码发行版本的 scripts 子目录里。找到这个文件后，在其目录里执行如下所示的命令：

```
% mysql -p -u root mysql < fill_help_tables.sql
```

上面这条命令将提示你输入 root 账户的口令。如果还没有为 root 账户设置口令，省略-p 选项即可

A.4.4 在Unxi系统上安装Perl DBI支持

如果你想编写一些能够访问 MySQL 数据库的 Perl 脚本，就需要安装 DBI 软件。

- 你必须安装提供各种DBI底层驱动程序的DBI模块，以及提供各种MySQL专用驱动程序的DBD::mysql模块。DBI需要Perl 5.6.0或更高的版本才能正常工作。（如果你的系统还没有安装Perl，请从http://www.perl.com/站点下载一个Perl发行版本并在安装DBI支持之前把它安装好。）
- 你的系统还必须安装有MySQL的C语言客户库和头文件，因为DBD::mysql模块需要用到它。作为MySQL软件的组件之一，这个C语言函数库应该已经安装好了。
- 如果你还想编写一些基于Web的DBI脚本，那就还得安装CGI.pm模块。

我们可以利用 perldoc 命令来查知某个 Perl 模块是否已经安装好了。如果已经安装了该模块，perldoc 命令就会把该模块的文档显示出来，如下所示：

```
% perldoc DBI
% perldoc DBD::mysql
% perldoc CGI
```

如果想把 Perl 模块安装到 Unix 系统上，最简单的办法是使用 CPAN shell。先以 root 用户登录，然后发出以下命令：

```
# perl -MCPAN -e shell
cpan> install DBI
cpan> force install DBD::mysql
cpan> install CGI
```

我们使用了 force install 命令来强行安装 DBD::mysql 模块，这样即使测试阶段失败了也会继续安装。所谓测试，是假设这几条命令可以使用一个没有口令的匿名 MySQL 用户账户连接到运行在本地主机上的 MySQL 服务器。但这么不安全的账户在实际工作中很难遇到，所以这个测试几乎总是会失败。（如果真的有一个这样的账户，那么即使省略了“force”测试也应该成功，但你无论如何都应该给所有的 MySQL 账户加上口令。）

安装 Perl 模块的另一个方法就是从 cpan.perl.org 站点下载 tar 压缩文件形式的源代码发行版本。你得先用下面两条命令之一来解压缩这个 dist_file.tar.gz 发行文件（如果你的 tar 命令不支持 z

选项，请使用第二条命令）：

```
% tar zxf dist_file.tar.gz
% gunzip < dist_file.tar.gz | tar xf -
```

然后，把当前路径切换到 tar 命令创建的那个子目录，再执行下面的命令（你可能需要以 root 用户身份来执行安装命令）：

```
% perl Makefile.PL
% make
% make test
# make install
```

正如刚解释过的，除非你有一个不安全的匿名 MySQL 用户账户，否则 make test 命令肯定会失败。如果想使用另外一个账户，请运行 perl makefile.PL 命令查看具体步骤。一般来说，用不着在意这个测试是否成功，继续执行其余的安装命令即可。

如果在安装 Perl 模块时遇到了麻烦，请查阅相关发行版本的 README 文件，或是到网上的 DBI 邮件列表寻求帮助，绝大多数与安装有关的问题都可以在那里找到答案。

A.4.5　在Unix系统上安装Apache和PHP

为 PHP 增添数据库访问功能的 PDO 扩展模块需要系统里安装了 PHP 5.0 或更高版本才能使用。如果使用的是 PHP 5.1 或更高版本，PDO 扩展模块已收录在了 PHP 发行版本里。如果使用的是 PHP 5.0，就需要另行安装 PDO，安装工作可以使用 pecl 命令行程序来完成。在 http://vwww.php.net/pdo 网站上可以查到更多信息。

在接下来的讨论里，我们将假设把 PHP 当做 Apache 服务器程序（httpd）的一个动态共享对象（DSO，dynamic shared object）来使用。这意味着应该先安装 Apache，然后才能构建和安装 PHP。如果系统还没有安装 Apache，可以找一个具备 DSO 支持功能的 Apache 二进制发行版本来直接安装，也可以在编译某个 Apache 源代码发行版本时把 DSO 支持包括进去并安装。

把 Apache 安装好以后，使用一条如下所示的命令（在同一行上输入）配置你的 PHP 发行版本。这条命令假设 Apache 和 MySQL 都安装在/usr/local 子目录下。请根据你的系统对其中的路径名作出调整。

```
% ./configure
    --with-apxs=/usr/local/apache/bin/apxs
    --enable-pdo
    --with-pdo-mysql=/usr/local/mysql
    --with-mysqli=/usr/local/mysql/bin/mysql_config
    --with-mysql=/usr/local/mysql
    --with-zlib
```

这条命令中用到的选项如下。对于每一个带有一个路径名值的选项，如果 configure 命令可以确定应该到哪里去查找必要的信息，在给出该选项时可以省略路径名。

- --with-apxs [= /path/to/apxs]
  告诉 configure 命令到哪里去寻找 apxs（Apache Extension Tool，Apache 扩展工具）的辅助脚本，该脚本用来向其他模块提供关于 Apache 配置情况的信息。
- --enable-pdo
  启用PHP的PDO支持机制。

- --with-pdo-mysql [= /path/to/mysql]
  带上 MySQL 支持的 PDO 扩展模块。路径名部分用来给出 MySQL 的安装位置，好让 configure 命令知道 MySQL 头文件和函数库的位置。
- --with-mysqli [= /path/to/mysq_config]
  带上 mysqli（意思是 MySQL improved，MySQL 改进版）扩展模块。这个选项并非使用 PDO 所必需的，其作用是让那些不喜欢使用 PDO 的人可以运行基于 mysqli 扩展模块的脚本。/path/to/mysq_config 是指向 mysq_config 脚本的路径名，configure 命令调用该脚本以获得 MySQL 配置信息。
- --with-mysql [= /path/to/mysql]
  带上 mysql 扩展模块。这个选项并非使用 PDO 所必需的。这个选项的作用是把 mysql 扩展模块构建到 PHP 里，让那些不喜欢使用 PDO 的人可以运行基于 mysql 扩展模块的脚本。路径名部分用来给出 MySQL 的安装位置，好让 configure 命令知道 MySQL 头文件和函数库的位置。
- --with-zlib
  这个选项必不可少。很多MySQL函数库都是经过压缩的。

配置好 PHP 发行版本后，用以下命令构建和安装它（你可能需要成为 root 用户才能执行这些安装命令）：

```
% make
# make install
# cp php.ini-dist /usr/local/lib/php.ini
```

上面这条 cp 命令将安装一个通用的 PHP 初始化文件到指定路径。如果你愿意，也可以把 php.ini-dist 文件替换为 php.ini-recommended 文件。我的建议是把这两个文件都查看一下，看哪一个更适用于你的系统。

把 PHP 和 PDO 都安装好以后，编辑 Apache 的配置文件 httpd.conf。你需要告诉 Apache 在它启动时应该去加载 PHP 模块和如何识别 PHP 脚本。（httpd.conf 文件可能还用 Include 指令导入了其他一些文件。如果在 httpd.conf 文件里没有看到下面段落里描述的信息，请到它导入的那些文件里去查看。）

为了让 Apache 加载 PHP 模块，需要在 httpd.conf 文件中的适当位置加上 LoadModule 和 AddModule 指令（也可能是其他类似的指令）。这些指令或许已经由 PHP 安装过程添加好了。如果不是这样，就必须由你本人添加。它们应该是下面这样，但 LoadModule 指令里的路径名可能需要根据系统的具体情况调整：

```
LoadModule php5_module libexec/libphp5.so
AddModule mod_php5.c
```

接下来，编辑 httpd.conf 文件告诉 Apache 如何识别 PHP 脚本。识别 PHP 脚本的依据是你为它们选用的文件扩展名。最常用的扩展名是“.php”，本书中的例子都将使用这个扩展名。如果你也决定使用“.php”作为 PHP 脚本的扩展名，请把下面这行代码添加到 httpd.conf 文件里去：

```
AddTtype application/x-httpd-php .php
```

还可以让 Apache 把 index.php 文件用作默认文件。如果用户提交的某个 URL 的末尾没有给出任何文件名，Apache 将使用它作为响应。http.conf 文件里会有一个如下所示的代码行：

```
DirectoryIndex index.html
```

请把它修改成下面这样：

```
DirectoryIndex index.php index.html
```

在编辑好 Apache 的配置文件以后，停止 httpd 服务器的运行（如果它正在运行），然后重新启动。在许多系统上，这可以用如下所示的命令来完成（使用 root 账户来执行）：

```
# /usr/local/apache/bin/apachectl stop
# /usr/local/apache/bin/apachectl start
```

也可以让 Apache 服务器随系统开机和关机而自动启动和停止（参见 Apache 文档）。一般来说，这是通过安排 apachectl start 命令在系统开机时运行、安排 apachectl stop 命令在系统关机时运行而实现的。

如果在安装 PHP 的过程中遇到了问题，在 PHP 发行版本中的 INSTALL 文件的“VERBOSE INSTALL”部分往往可以找到解决办法。（INSTALL 文件包含许多有用的信息，应该提前研读以防万一。）

A.5　在 Windows 系统上安装 MySQL

本节内容需要假设你已经安装了 Windows 2000、XP、2003 或 Vista 等比较新的版本。本书里讨论的某些功能，如命名管道和 Windows 服务，在老版本（Windows 95、Windows 98 或 Windows Me）里是没有的。

适用于 Windows 系统的 MySQL 发行版本主要有 3 种。

- zip压缩文件形式的免安装发行版本，里面包含安装MySQL软件所需要的所有组件。这种免安装发行版本的文件名以“mysql-noinstall”开头，只需要把压缩文件解压缩到一个文件夹，再把该文件夹移动到你打算安装MySQL软件的位置，就可以完成安装工作。比如说，解压缩mysql-noinstall-5.0.51-win32.zip文件将生成一个名为mysql-noinstall-5.0.51-win32的文件夹，如果打算把MySQL安装到C:\mysql子目录，把该文件夹重新命名为mysql并把它移动到C:盘的根目录下即可。
- 完整安装包，里面包含一个Configuration Wizard（配置向导）和安装MySQL软件所需要的所有组件。这种完整安装包有着mysql-*version*-win32.zip形式的文件名。完整安装包的大致安装过程是：下载它，执行其中包含的Setup.exe程序，然后按照对话框里的指示继续进行。
- 基本安装包，类似于完整版，但只包含安装MySQL软件所需要的最少文件。比如说，它省略了MySQL服务器的调试版本。每个基本安装包就是一个自安装程序，文件名以“mysql-essential”开头，扩展名是“.msi”。基本安装包的大致安装过程是：下载它，执行它，然后按照对话框里的指示继续进行。

完整安装包和基本安装包都喜欢把 MySQL 软件安装到 C:\Program Files\MySQL 文件夹而不是 C:\mysql 文件夹。它们还会在 Windows 的“开始”菜单里创建一个快捷方式，在 Windows 注册表里留下一些注册项，但免安装发行版本不会这样做。

关于在 Windows 系统上安装 MySQL 软件的更多信息，请参阅《MySQL 参考手册》中的安装章节。

如果想从命令行启动 MySQL 程序而无需输入完整的路径名，需要把 MySQL 安装目录中的 bin 子目录包括到 PATH 环境变量里。比如说，如果把 MySQL 软件安装在 C:\mysql 文件夹里，就需要把“C:\mysql\bin”添加到 PATH 环境变量里去。可以通过 Windows“控制面板”中的“系统”页面来设置路径，设置完成后通常还需要重新启动 Windows 才能让改动生效。

接下来的讨论将假设已经设置好了 PATH 变量，这样在输入命令时就不必输入它们的路径了（注意，mysql --install 命令是个例外，你必须给出 mysqld 程序的完整路径）。

一般来说，在 Windows 系统上安装好 MySQL 软件之后，用不着对 MySQL 数据目录或者权限数据表进行初始化，因为它们在发行版本里通常都已经接受过预初始化了。不过，万一没有把 MySQL 软件安装到其安装程序默认选定的位置，就必须往 MySQL 服务器在启动时会读取的某个选项文件里加上 [mysqld]选项组，确保 MySQL 服务器程序能够找到 MySQL 安装目录和数据目录。最常用的选项文件是 MySQL 安装目录里的 my.ini 文件和 C:\my.ini 文件。比如说，如果把 MySQL 安装在 C:\mysql，就应该增加一个如下所示的选项组（请注意，路径名里使用的是斜线字符“/”而不是反斜线字符“\”）：

```
[mysqld]
basedir=C:/mysql
datadir=C:/mysql/data
```

如果使用了一个不同的安装目录，千万不要忘记对选项文件里的路径名作相应的修改。

适用于 Windows 的 MySQL 发行版本通常都会收录好几种 MySQL 服务器程序，它们是使用不同的选项构建作出来的。

- 在MySQL 5.1.12版之前，mysqld-nt和mysqld是两种不同的服务器程序，前者支持命名管道，后者不支持。
- 从MySQL 5.1.12版开始，mysqld全面支持命名管道，mysqld-nt则不复存在。
- 所有版本里的mysql-debug服务器程序都支持命名管道、调试功能、自动化的内存分配检查功能等。

一般来说，除非需要用到 mysql-debug 服务器程序所提供的调试功能，否则最好是选择另一种服务器程序。与 Windows 平台上的其他几种 MySQL 服务器程序相比，mysql-debug 将占用更多的内存，运行速度却慢得多。

有好几种服务器程序都支持使用命名管道的连接，但这种连接在默认情况下是禁用的。所有服务器都支持共享内存连接，但这些在默认情况下也是禁用的。如果想启用这些能力，就必须在选项文件的[mysqld]选项组里增加合适的设置行：

```
[mysqld]
enable-named-pipe
shared-memory
```

在 Windows 平台上，安装的任何一种 MySQL 服务器程序都可以作为一项 Windows 服务，在 Windows 启动时自动运行。比如说，下面这条命令将把 mysqld 服务器程序安装为一项 Windows 服务：

```
C:\> C:\mysql\bin\mysqld --install
```

--install 命令要求给出服务器程序的完整路径名。如果把服务器程序安装在了其他位置，请对路径名进行必要的修改。

如果使用的是--install-manual 命令而不是--install 命令，服务器程序仍将被安装为一项 Windows 服务，但不会在 Windows 启动时自动运行。你必须通过 Windiws Service Manager（Windows 的服务管理器）工具或 net start 命令才能让它运行起来。

如果打算把 MySQL 服务器安装为一项 Windows 服务，可以把有关选项放在某个选项文件中的[mysqld]选项组里。

如果已经把 MySQL 服务器安装为一项 Windows 服务，就可以通过 Windiws Service Manager 工具

手动启动它。应该可以在 Windows Control Panel（即 Windows 的控制面板）中的 Services（服务）或 Administrative Tools（管理工具）页面里找到这项服务。该服务还可以使用如下所示的命令来启动：

```
C:\> net start MySQL
```

服务器可以用 Service Manager 或者下列命令之一来停止：

```
C:\> net stop MySQL
C:\> mysqladmin -u root shutdown
```

如果想删除被安装为一项 Windows 服务的 MySQL 服务器，先关闭该服务器（如果它正在运行），然后执行下面的命令：

```
C:\> mysqld --remove
```

为了避免 Service Manager 和命令提示符相互干扰，在你从命令行发出与 Windows 服务有关的命令时最好先关闭 Service Manager。

如果没有把 MySQL 服务器安装为一项服务，可以从命令行以手动方式来启动或停止该服务器。比如说，下面这条命令将启动 mysqld 服务器：

```
C:\> mysqld
```

如果愿意，你可以在命令行上给出其他选项。如果想关闭服务器，可以使用 mysqladmin 工具：

```
C:\> mysqladmin -u root shutdown
```

如果想让 MySQL 服务器以控制台模式运行——这将使它把出错信息显示在一个控制台窗口里，你可以使用--console 选项来启动它。比如说，下面这条命令将让 mysqld 服务器程序以控制台模式运行：

```
C:\> mysqld --console
```

在以控制台模式运行 MySQL 服务器时，可以把其他选项列在命令行上的--console 选项后面，也可以把它们列在某个选项文件里。服务器可以用 mysqladmin 工具来关闭。

注意　*从命令行启动 MySQL 服务器之后，在它退出运行之前你不一定能看到一个新的命令提示符。如果是这样，只要再打开一个控制台窗口就可以运行 MySQL 客户程序了。*

如果在运行 MySQL 服务器时遇到了麻烦，请参阅《MySQL 参考手册》中有关安装的章节中介绍的 Windows 部分。

如果是首次安装 MySQL 软件，还有几件事是应该趁现在做好的。

- 默认的安装设置通常会允许任何人使用MySQL的root账户而无需输入任何口令。出于安全方面的考虑，应该给它们设置好口令。
- 可以安排MySQL服务器随着系统的正常开机和关机而自动地启动和关停。
- 可以把--user选项放在某个选项文件里，这样就用不着每次启动MySQL服务器都输入它了。
- 可以趁现在创建多个不同用途的登录账户，如用来监视MySQL服务器运行情况的、用来完成复制工作的、用来完成数据备份和恢复工作的，等等。
- 可以启用或禁用存储引擎，或者为它们设定必要的优化参数。

完成以上各项任务的具体操作步骤在第 12 章里已详细描述过。

如果是对已安装的现有 MySQL 系统升级，权限数据表的结构有可能已经发生了变化。如果需要

升级那些数据表，使用最新的结构，请按照 12.4 节的步骤操作。

适用于 Windows 平台的当前 MySQL 发行版本里，为 `mysql` 客户程序的 `help` 命令而准备的“帮助”数据表都已经设置到位，应该不需要再手动创建和加载它们了。不过，为支持 MySQL 用户使用地理时区的名字来设置时区而准备的“地理时区”数据表却仍有可能是缺失或空白的。如果需要对“地理时区”数据表设置，请按照 12.9.1 节的步骤操作。

A.5.1 在Windows系统上安装Perl DBI支持

在 Windows 系统上安装 Perl 模块的最简单办法是先从 www.activestate.com 网站下载 ActiveState Perl 发行版本并安装，然后再根据具体需要去下载和安装其他的 Perl 模块。ppm（Perl Package Manager，Perl 软件包管理器）程序可以用来完成这项任务：

```
C:\> ppm
ppm> install DBI
ppm> install DBD-mysql
ppm> install CGI
```

A.5.2 在Windows系统上安装Apache和PHP

在 A.2 节中列举的 Apache 和 PHP 网站上可以找到适用于 Windows 平台的 Apache 和 PHP 二进制发行版本。在 Apache 2.x 环境下，既可以把 PHP 作为一个独立程序运行，也可以作为一个 Apache 模块运行。

为 PHP 增添数据库访问功能的 PDO 扩展模块需要系统里安装了 PHP 5.0 或更高版本才能使用。适用于 Windows 平台的 PHP 二进制发行版本有 Zip 文件和.msi 文件两种格式。如果使用的是 Zip 文件，在你想安装 PHP 的文件夹里对它解压缩即可。.msi 文件形式的安装包更实用一些，因为它可以一步一步地引导你安装 PHP、配置 Apache 对 PHP 的支持功能、设置 `PATH` 变量包括 `PHP` 安装位置。但如果使用的是.msi 文件，一定要选择“扩展安装”，否则 PDO 和 MySQL 支持将不会被安装。

附录 B

数据类型指南

本附录将介绍 MySQL 提供的各种数据列类型。各种数据列类型的使用方法在本书第 3 章有详细的讨论。从 MySQL 5.0.0 版起各个类型的行为变化将在这里指出。

在本附录中的类型名定义里，我们使用了以下几种记法。

- 方括号（[]）。可选信息。
- M代表整数类型的最大显示宽度、浮点类型和DECIMAL类型的精度（有效数字的个数）、BIT类型的二进制位的个数、字符串类型的最大长度。在字符串数据列的定义里，二进制字符串的长度单位是字节，非二进制字符串的长度单位是字符。
- D代表带有小数部分的类型的小数位数（小数点后面的数字的个数），也叫做数值范围。D不应该大于M。从MySQL 5.0.10版开始，M不得小于D，否则将出错。在MySQL 5.0.10之前的版本里，如果M小于D，则M的值将调整为D+1。

在介绍每一种数据列类型的时候，我们将给出以下某方面或某几个方面的信息。

含义——对该类型的简短描述。

可用属性——能够通过 CREATE TABLE 或 ALTER TABLE 语句有选择地加在该数据列类型上的属性关键字。这些属性将按字母表顺序依次列出，但这不表示你在 CREATE TABLE 或 ALTER TABLE 语句里也必须按同样的顺序列出这些属性。CREATE TABLE 和 ALTER TABLE 语句的语法请参阅附录 E。本附录各数据列类型条目里列出来的属性是应用于全部或几乎全部数据类型的全局属性的补充。

- Null 或 NOT NULL 应为每种类型指定。
- DEFAULT default_value 应指定给 BLOB 和 TEXT 列、坐标列、除有 AUTO_INCREMENT 属性的整数列以外的所有列。除 TIMESTAMP 类型外，默认值必须是常数。例如，你不能为 DATE 列指定 DEFAULT CORDATA()。

最大长度。适用于字符串类型，它指的是该类型的数据列值被允许达到的最大长度。

表示范围。适用于数值类型和日期/时间类型，它指的是该类型所能表示的最大值。整数类型因为存在带正负符号和不带正负符号两种情况，所以都有两个表示范围，并且不同情况有不同的表示范围。

零值。日期/时间类型的“零”值指的是 MySQL 在你试图把一个非法值插入到该类型的数据列里时真正放入数据列里的值。（SQL 模式中，应设置为允许如此，或者插入非法值时，有错误发生。）

默认值。类型定义没有明确设定 DEFAULT 属性时的默认值。这只在没有启用严格 SQL 模式时才适用。如果在严格模式中没有给出 DEFAULT 子句，并且能接受 NULL 值，那么就会用默认的 NULL 来定义列，否则，用非默认值定义。参见 3.2.3 节可了解详情。

存储空间占用量。存储有关类型的数据值所需要使用的字节个数。有些类型的存储空间占用量是

一个固定的数字，有些类型的存储空间占用量却是一个可变的数字，要由存入数据列里的数据值的长度来决定。

比较方式。适用于字符串类型，用来表明该类型上比较操作的方式。这对分组、排序和索引操作也有影响，因为这两种操作都是在比较操作的基础上进行的。二进制字符串类型是使用每种类型的数字值逐类型地比较的。非二进制字符串类型是根据字符集排列顺序逐字符地比较的。

同义词。类型名称的同义词。

其他事项。有关类型需要注意的其他特点。

备注：有关该类型的额外说明。

如果你不确定自己的 MySQL 版本将如何对待给定列的定义，这里有一个提示。创建一个表，包含一个列，其定义方式是你想要知道的。然后使用 SHOW CREATE TABLE 或 DESCRIBE 来查看 MySQL 如何报告一个定义。例如，如果记不起 UNICODE 字符类型属性或 SERIAL 缩写数据类型的作用，那么创建一个表来使用它们，然后告诉 MySQL 显示结果表的定义。

```
mysql> CREATE TABLE t (c CHAR(10) UNICODE, s SERIAL);
mysql> SHOW CREATE TABLE t\G
*************************** 1. row ***************************
       Table: t
Create Table: CREATE TABLE `t` (
  `c` char(10) character set ucs2 default NULL,
  `s` bigint(20) unsigned NOT NULL auto_increment,
  UNIQUE KEY `s` (`s`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1
```

B.1 数值类型

MySQL 提供了精确值和近似值两大类数据类型。不同数值类型有着不同的取值范围，应该根据想要表示的数值的范围来加以选择。还有一种 BIT 类型，专门用来表示二进制值。

整数和定点数（DECIMAL）类型是精确值数据类型。FLOAT 和 DOUBLE 类型则是近似值数据类型。对于精确值类型，数值在存储和使用时是完全一致的，只要数值和计算结果没有超出这些类型的取值范围，计算过程就将精确地进行而不会产生任何舍入误差。精确值类型与硬件平台无关，使用它们获得的结果在所有的系统上都是相同的。对于近似值类型，因为浮点操作在不同硬件平台上的实现难免会有一些细微的差异，计算过程也就难免产生舍入错误和误差。

对于整数类型的数据列，只要给它加上了 AUTO_INCREMENT 属性，就必须对它编写索引。把 NULL 值插入一个 AUTO_INCREMENT 列将导致下一个序列值被插入到该列里。最常见的情况是新序列值等于该数据列里的当前最大值加上 1。第 3 章对 AUTO_INCREMENT 数据列的精确行为作了详细的描述。（实际上，AUTO_INCREMENT 属性也可以用于浮点数据类型，只是它与整数列搭配使用的情况更为常见而已。）

ZEROFILL 和 UNSIGNED 属性可以用于除 BIT 类型以外的所有数值类型。

- 如果给某个数据列加上了ZEROFILL属性，MySQL在显示该数据列时会用一些前导的零把它们“扩展”到该数据列的最大显示宽度。
- 如果给某个数据列加上了UNSIGNED属性，该数据列将不允许出现负值。（SIGNED其实也是一种可用属性，但因为数值类型在默认情况下都带有正负号，所以没有什么实际效果。）

适用于整数和浮点数据类型的 SERIAL DEFAULT VALUE 属性是 NOT NULL AUTO_INCREMENT

UNIQUE 的缩写形式。

在某些场合，即使你只设定了一种属性，也会导致其他几种属性也被启用。给数值类型设定 ZEROFILL 属性的做法将使有关数据列被自动地设定为 UNSIGNED。只要设定了 AUTO_INCREMENT 属性，有关数据列就将被自动设定为 NOT NULL。

虽然 DESCRIBE 和 SHOW COLUMNS 命令会报告说某个 AUTO_INCREMENT 数据列的默认值是 NULL，但你却无法真的把一个 NULL 值插入到这个数据列里去。它只表明这样一个事实：如果你在创建一个新行时给 AUTO_INCREMENT 数据列赋了一个 NULL 值，就会在该数据列里产生一个默认值（即下一个序列编号）来。

B.1.1　整型

- TINYINT [(*M*)]

  含义：一个非常小的整数。*M* 是最大显示宽度，从 1 到 255。如果省略，*M* 的默认值是 4（对于无符号数据列，这个默认值是 3）。

  可用属性：AUTO_INCREMENT、SERIAL、DEFAULT VALUE、UNSIGNED、ZEROFILL

  表示范围：带符号，−128 到 127（-2^7 到 2^7-1）；无符号，0 到 255（0 到 2^8-1）

  默认值：如果数据列允许使用 NULL 值，默认值为 NULL；如果带 NOT NULL 属性，默认值为 0。

  存储空间占用量：1 个字节。

  同义词：INT1 [(*M*)]。此外，BOOL 和 BOOL EAN 是 TINYINT(1) 的同义词。在 MySQL 5.0.3 之前，BIT 是 TINYINY(1)的同义词。从 5.0.3 起，BIT 是一种单独的数据类型。

- SMALLINT [(*M*)]

  含义：一个小整数。*M* 是最大显示宽度，从 1 到 255。如果省略，*M* 的默认值是 6（对于无符号数据列，这个默认值是 5）。

  可用属性：AUTO_INCREMENT、SERIAL DEFAULT VALUE、UNSIGNED、ZEROFILL

  表示范围：带符号，−32768 到 32767（-2^{15} 到 $2^{15}-1$）；无符号，0 到 65535（0 到 $2^{16}-1$）

  默认值：如果数据列允许使用 NULL 值，默认值为 NULL；如果带 NOT NULL 属性，默认值为 0。

  存储空间占用量：2 个字节。

  同义词：INT2 [(*M*)]

- MEDIUMINT [(*M*)]

  含义：一个中等尺寸的整数。*M* 是最大显示宽度，从 1 到 255。如果省略，*M* 的默认值是 9（对于无符号数据列，这个默认值是 8）。

  可用属性：AUTO_INCREMENT、SERIAL DEFAULT VALUE、UNSIGNED、ZEROFILL

  表示范围：带符号，−8388608 到 8388607（-2^{23} 到 $2^{23}-1$）；无符号，0 到 16777215（0 到 $2^{24}-1$）

  默认值：如果数据列允许使用 NULL 值，默认值为 NULL；如果带 NOT NULL 属性，默认值为 0。

  存储空间占用量：3 个字节。

  同义词：INT3 [(*M*)]和 MIDDLENT [(*M*)]

- INT [(*M*)]

  含义：一个标准的整数。*M* 是最大显示宽度，从 1 到 255。如果省略，*M* 的默认值是 11（对于无符号数据列，这个默认值是 10）。

  可用属性：AUTO_INCREMENT、SERIAL DEFAULT VALUE、UNSIGNED、ZEROFILL

表示范围：带符号，−2147483648 到 2147483647（-2^{31} 到 $2^{31}-1$）；无符号：0 到 4294967295（0 到 $2^{32}-1$）

默认值：如果数据列允许使用 NULL 值，默认值为 NULL；如果带 NOT NULL 属性，默认值为 0。

存储空间占用量：4 个字节。

同义词：INTERGER [(*M*)]和 INT4 [(*M*)]

- BIGINT [(*M*)]

  含义：一个大整数。*M* 是最大显示宽度，从 1 到 255。如果省略，*M* 的默认值是 20。

  可用属性：AUTO_INCREMENT、SERIAL DEFAULT VALUE、UNSIGNED、ZEROFILL

  表示范围：带符号，−9223372036854775808 到 9223372036854775807（-2^{63} 到 $2^{63}-1$）

  无符号：0 到 18446744073709551615（0 到 $2^{64}-1$）

  默认值：如果数据列允许使用 NULL 值，默认值为 NULL；如果带 NOT NULL 属性，默认值为 0。

  存储空间占用量：8 个字节。

  同义词：INT8 [(*M*)]

  其他事项：作为一种数据类型，SERIAL 是 BIGINT UNSIGNED NOT NULL AUTU_INCREMENT UNIQUE 的简写。

B.1.2 定点数据类型

- DECIMAL[(*M*, [*D*])]

  含义：一个定点数。*M* 是有效数字的个数，从 0 到 65，*D* 是小数点后面的精确位数，从 0 到 30。如果 *D* 等于 0，数据将没有小数点或小数部分。如果 *M* 和 *D* 省略，*M* 和 *D* 将被默认地设置为 10 和 0。

  可用属性：UNSIGNED、ZEROFILL

  表示范围：给定的 DECIMAL 列的范围由 *M* 和 *D* 的值以及是否给定 UNSIGNED 属性来决定。

  默认值：如果数据列允许使用 NULL 值，默认值为 NULL；如果带 NOT NULL 属性，默认值为 0。

  存储空间占用量：取决于小数点左边和右边的数字的总个数。在小数点的两边，每 9 个数字占用 4 个字节，再加上剩余部分可能占用的 1 到 4 个字节。每个数值的存储空间等于小数点左、右两边字节占用量的和。

  同义词：NUMERIC [(*M*, [*D*])]、DEC [(*M*, [*D*])]、FIXED [(*M*, [*D*])]

  备注：在 MySQL 5.0.3 之前的版本里，DECIMAL 值被存储为字符串，因而在某些方面与现如今的表示方式有所不同。详见《MySQL 参考手册》。

B.1.3 浮点型

- FLOAT(*p*)

  含义：一个浮点数。在标准 SQL 中，精确度 *p* 表示需要最少位数。在 MySQL 中，*p* 仅用于确定数据类型是单精度还是双精度的。

  - 如果 *p* 值在 0 到 24 中间，浮点值将被看做是单精度浮点数，即相当于不带 *M* 和 *D* 参数的 FLOAT 类型。
  - 如果 *p* 值在 25 到 53 之间，浮点值将被看做是双精度浮点数，即相当于不带 *M* 和 *D* 参数的 DOUBLE 类型。

不在 0 与 53 之间的 *p* 值是非法。

可用属性：UNSIGNED、ZEROFILL

表示范围：参见后面对 FLOAT 和 DOUBLE 类型的描述。

默认值：如果数据列允许使用 NULL 值，默认值为 NULL；如果带 NOT NULL 属性，默认值为 0。

存储空间占用量：单精度值需要 4 个字节，双精度值需要 8 个字节。

- FLOAT [(*M*, *D*)]

  含义：一个单精度浮点数（精度小于 DOUBLE 类型）。*M* 是有效位数，从 0 到 255，*D* 是小数点后面的精确位数，从 0 到 30。如果 *D* 等于 0，数据将没有小数点或小数部分。如果 *M* 和 *D* 都被省略，显示宽度和小数精度将不确定。

  可用属性：UNSIGNED、ZEROFILL

  表示范围：最小非零值为±1.175494351E−38，最大非零值为±3.402823466E+38。如果给一个浮点数据列加上 UNSIGNED 属性，它就不允许再容纳负值。

  默认值：如果数据列允许使用 NULL 值，默认值为 NULL。如果带 NOT NULL 属性，默认值为 0。

  存储空间占用量：4 个字节。

  同义词：FLOAT4 是不带 *M* 和 *D* 时的 FLOAT 的同义词。如果 REAL_AS_FLOAT 模式启用了，在 MySQL 3.23.6 之前的版本里，REAL[(*M*, *D*)]是 FLOAT(*M*, *D*)的同义词。

- DOUBLE [(*M*, *D*)]

  含义：一个双精度浮点数（精度大于 FLOAT 类型）。*M* 和 *D* 的含义与在 FLOAT 中时一样。

  可用属性：UNSIGNED、ZEROFILL

  表示范围：最小非零值是±2.2250738585072014E−308，最大非零值是±1.7976931348623157E+308。如果给一个浮点数据列加上 UNSIGNED 属性，它就不允许再容纳负值。

  默认值：如果数据列允许使用 NULL 值，默认值为 NULL。如果带 NOT NULL 属性，默认值为 0。

  存储空间占用量：8 个字节。

  同义词：DOUBLE PRECISION [(*M*, *D*)]和 REAL [(*M*, *D*)]（如果没启用 REAL_AS_FLOAT）是 DOUBLE [(*M*, *D*)]的同义词。FLOAT8 是 DOUBLE 不带 *M* 或 *D* 时的同义词。

B.1.4 BIT类型

BIT [(*M*)]

含义：一个位字段值。*M* 是 1 到 64 之间的一个整数，用来表明每个 BIT 值有多少位。如果省略，*M* 的默认值是 1。

可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

默认值：如果数据列允许使用 NULL 值，默认值为 NULL；如果带 NOT NULL 属性，默认值为 0。

存储空间占用量：大约(*M*+ 7)/8 个字节。

其他事项：BIT 类型从 MySQL 5.0.3 版开始才成为一种独立的数据类型。最早的 BIT 支持仅限于 MyISAM，在 MySQL 5.0.5 版里扩展到 InnoDB、MEMORY 和 ARCHIVE；在 MySQL 6.0 里扩大到 Falcon。在 MySQL 5.0.3 版之前，BIT 是 TINYINT(1)的一个同义词。

B.2 字符串类型

MySQL 中的字符串类型是通用类型，通常用来存储二进制或者字符（文本）数据。字符串类型

有很多种，可以根据字符串值的最大长度来选择，也可以根据你想把它们视为二进制字符串还是非二进制字符串来选择。

BINARY、VARBINARY 和 BLOB 类型是二进制字符串类型。每个二进制字符串是一个字节序列，其长度单位是字节。二进制字符串没有字符集的概念，比较操作以字节值为基础，实际上是依次比较每组对应字节的数值。

CHAR、VARCHAR 和 TEXT 类型是非二进制字符串类型。每个非二进制字符串是一个字符序列，它有字符集和排序方式的概念。字符集为数据类型定义允许使用的字符，排序方式为之定义字符排序顺序。你在一个非二进制字符串数据列的定义里给出的长度表明了你想让这个数据列能够容纳多少个字符。

非二进制字符串值的长度通常以字符为单位，但也可以使用字节作为长度单位。若想获得某个非二进制字符串以字符或字节计算的长度，分别使用 CHAR_LENGHT()或 LENGTH()函数即可。如果某个非二进制字符串只包含单字节字符，它按字符计算的长度和按字节计算的长度将是一样的；如果它包含多字节字符，其字符长度将小于字节长度。这种差异会影响非二进制字符串数据列的存储空间占用量。

- 固定长度的数据列（如CHAR(*M*)）所需要的存储空间必须足以让它存储*M*个给定字符集里最宽字符。比如说，utf8字符集里的每个字符需要占用1到3个字节，所以CHAR(*M*)需要*M* × 3个字节。（在MySQL 6.0.4和更高的版本里，utf8字符最多需要4个字节。）
- 可变长度的数据列（如VARCHAR(*M*)）所需要的存储空间只要足以存储一个给定字符串的实际字符、再加上几个用来存储该字符串的字节长度的前缀字节即可。一个使用双字节ucs2字符集的VARCHAR(10)数据列需要0（空字符串）到20个字节（10个字符）来存储字符串里的字符，再加上一个字节作为其长度前缀。

你可以为非二进制字符串类型（CHAR、VARCHAR、TEXT）、ENUM 和 SET 类型指定一个字符集和排序方式。

- 字符集的应用语法是CHARACTER SET charset，其中charset是某个字符集的名字，如latin1、greek或utf8。CHARSET是CHARACTER SET的一个同义词。
- 排序方式的设定语法是COLLATE collation，其中是collation是给定字符集的可用排序方式之一的名字。
- 如果你既没有设定字符集，也没有设定排序方式，MySQL将根据数据表的默认设置来确定。如果只设定了字符集而没有设定排序方式，MySQL将使用该字符集的默认排序方式。如果只设定了排序方式而没有设定字符集，MySQL将根据排序方式的名字来确定字符集。如果既设定了字符集又设定了排序方式，排序方式必须和字符集相兼容。比如说，latin1_bin排序方式和latin1字符集兼容，但和utf8字符集不兼容。
- binary字符集和BINARY列属性是区别对待的。
  - 如果为某种非二进制字符串类型指定了CHARACTER SET binary字符集，该类型将被转换为相对应的二进制字符串类型，即CHAR类型将变成BINARY类型，VARCHAR类型将变成VARBINARY类型，TEXT类型将变成BLOB类型。ENUM和SET类型没有相对应的二进制类型，CHARACTER SET binary在遇到ENUM和SET类型的时候将变成一个数据列属性。
  - BINARY属性相当于在指定了字符集（以_bin结尾的排序名称）之后再指定其二进制排序方式。比如说，一个被定义为CHAR(10) CHARACTER SET utf8 BANARY的数据列将变成CHARACTER SET utf8 COLLATE utf8_bin。

- 对非二进制字符串类型而言，ASCII和UNICODE属性分别是CHARACTER SET latin1和CHARACTER SET ucs2的缩写。

服务器支持的字符集和排序方式可以通过 SHOW CHARACTER SET 和 SHOW COLLATION 语句查出。这些语句将报告每种可用字符集的默认排序方式。你还可以去查询 INFORMATION_SCHEMA 数据库里的 CHARACTER_SETS 和 COLLATIONS 数据表，它们包含同样的信息。

有些长到无法存储在字符串数据列里的值，对这些值的处理取决于 SQL 模式设置。如果没有启用“严格”模式，超长的值将被截短到刚好能够容纳在该数据列里；如果被截去的字符不全是空格，MySQL 还将生成一条警告消息。在“严格”模式下，系统将报告出错，如果还有非空格字符必须被截去的话，MySQL 将不会向数据列插入任何值。

对尾缀值的处理分为以下几种情况。

- 对于CHAR类型，如果必要MySQL在存储字符串值时会使用尾缀空格补齐到数据列的长度，在检索时会去掉尾缀的空格。
- 对于BINARY类型，如果必要MySQL在存储字符串值时会使用零值（0x00）字节补齐到数据列的长度，但在检索时不会去掉尾缀的零值字节。（在MySQL 5.0.15版之前，MySQL在存储BINARY值时会使用空格来补齐长度，在检索时会去掉尾缀的空格。）
- 对于VARBINARY和VARCHAR类型，MySQL在存储或检索字符串值时不进行任何补齐或截短处理。（在MySQL 5.0.3版之前，MySQL会在存储这两类值时先去掉它们的尾缀空格。）
- 对于BLOB和TEXT类型，MySQL在存储或检索字符串值时不进行任何补齐或截短处理。
- 对于ENUM和SET类型，在数据列定义里列出的成员值的任何尾缀空格都将被忽略。因此，存储在数据列里的值没有任何尾缀的空格，因为MySQL会把每个值转换为对应于该数据列成员的内部数值。这同样会影响到比较操作，因为在对ENUM或SET数据列里的值进行比较的时候，尾缀的空格不参加比较。

B.2.1　二进制字符串类型

- BINARY [(*M*)]

  含义：一个固定长度的二进制字符串，长度在 0 到 *M* 个字节之间。*M* 应该是 0 到 255 之间的一个整数。如果省略，*M* 的默认值是 1。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  允许长度：0 到 *M* 个字节。

  默认值：如果数据列允许为 NULL，则默认值为 NULL；如果为 NOT NULL，默认值为空字符串（''）。

  存储空间占用量：*M* 个字节。

  比较方式：依次比较各个字节的数值。

- VARBINARY [(*M*)]

  含义：一个可变长度的二进制字符串，长度在 0 到 *M* 个字节之间。*M* 应该是 0 到 65535 之间（在 MySQL 5.0.3 版之前是 0 到 255 之间）的一个整数。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  允许长度：0 到 *M* 个字节。

  默认值：如果数据列允许为 NULL，则默认值为 NULL；如果允许 NOT NULL，默认值为空字符

串（''）。

存储空间占用量：数据本身的长度（以字节计算）再加上1或2个字节的前缀。如果数据列值的最大长度小于256个字节，需要1个字节的前缀；否则，需要2个字节。

比较方式：依次比较各个字节的数值。

其他事项：在实际工作中，因为各存储引擎在其内部对数据行的最大长度有一个上限，而且数据表里的其他数据列也要占用空间，所以一个VARBINARY数据列的实际最大长度通常都小于655 35个字节。

- TINYBLOB

  含义：一个小尺寸的BLOB（二进制字符串）值。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  允许长度：0到255（0到2^8-1）个字节。

  默认值：如果数据列允许为NULL，则默认值为NULL；如果允许为NOT NULL，默认值为空字符串（''）。

  存储空间占用量：数据本身的长度（以字节计算）再加上1个用来保存这个长度值的字节。

  比较方式：依次比较各个字节的数值。

- BLOB [(*M*)]

  含义：一个标准尺寸的BLOB（二进制字符串）值。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  允许长度：0到65535（0到$2^{16}-1$）个字节。如果给出了长度*M*，*M*将被用来选择一种适当的数据类型，然后被丢弃。如果*M*的值在1到65535之间，数据类型变成BLOB；如果*M*的值等于或大于65535，数据类型将根据数据值的实际长度变成MEDIUMBLOB或LONGBLOB。

  默认值：如果数据列允许为NULL，则默认值为NULL；如果允许为NOT NULL，默认值为空字符串（''）。

  存储空间占用量：数据本身的长度（以字节计算），再加上2个用来保存这个长度值的字节。

  比较方式：依次比较各字节的数值。

- MEDIUMBLOB

  含义：一个中等尺寸的BLOB（二进制字符串）值。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  允许长度：0到16777215（0到$2^{24}-1$）个字节。

  默认值：如果数据列允许为NULL，则默认值为NULL；如果允许为NOT NULL，默认值为空字符串（''）。

  存储空间占用量：数据本身的长度（以字节计算），再加上3个用来保存这个长度值的字节。

  比较方式：依次比较各个字节的数值。

  同义词：LONG VARBINARY。

- LONGBLOB

  含义：一个大尺寸的BLOB（二进制字符串）值。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  允许长度：0到4294967295（0到$2^{32}-1$）个字节。

  默认值：如果数据列允许为NULL，则默认值为NULL；如果允许为NOT NULL，默认值为空字

符串（''）。

存储空间占用量：数据本身的长度（以字节计算），再加上 4 个用来保存这个长度值的字节。

比较方式：依次比较各个字节的数值。

B.2.2 非二进制字符串类型

- CHAR [(*M*)]

含义：一个固定长度的非二进制字符串，长度在 0 到 *M* 个字符之间。*M* 应该是 0 到 255 之间的一个整数。如果省略，*M* 的默认值是 1。

可用属性：BINARY、CHARACTER SET、COLLATE

允许长度：0 到 *M* 个字符。

默认值：如果数据列允许为 NULL，则默认值为 NULL；如果允许为 NOT NULL，默认值为空字符串（''）。

存储空间占用量：*M* 个字符，长度是 *M***w* 个字节，*w* 是数据列所使用的字符集里的最宽字符需要占用的字节个数。

比较方式：根据数据列所使用的排序方式依次比较各个字符。

同义词：NCHAR(*M*)和 NATIONAL CHAR(*M*)是 CHAR(*M*) CHARACTER SET utf8 的同义词。

- VARCHAR (*M*)

含义：一个可变长度的非二进制字符串，长度在 0 到 *M* 个字符之间。*M* 应该是 0 到 65536 之间（在 MySQL 5.0.3 版之前是 0 到 255 之间）的一个整数。

可用属性：BINARY、CHARACTER SET、COLLATE

允许长度：0 到 *M* 个字符。另见“其他事项”中的说明。

默认值：如果数据列允许为 NULL，则默认值为 NULL；如果允许为 NOT NULL，默认值为空字符串（''）。

存储空间占用量：数据本身的长度（以字节计算），再加上 1 或 2 个字节的前缀，用来保存数据本身的长度。如果数据列值的最大长度小于 256 个字节，需要 1 个字节的前缀；否则，需要 2 个字节，

比较方式：根据数据列所使用的排序方式依次比较各个字符。

同义词：CHAR VARYING(*M*)、NVRACHAR(*M*)、NCHAR VARYING(*M*)和 NATIONAL CHAR VARYING(*M*)是 VARCHAR(*M*) CHARACTER SET utf8 的同义词。

其他事项：在实际中，一个 VARCHAR 数据列的最大长度不能超过 65 535 个字节。此外，因为各存储引擎在其内部对数据行的最大长度有一个上限，数据列所使用的字符集可以是单字节字符集或多字节字符集，数据表里的其他数据列也要占用空间等原因，所以一个 VARCHAR 数据列的实际最大长度通常都小于 *M* 个字符。

- TINYTEXT

含义：一个小尺寸的TEXT值。

可用属性：BINARY、CHARACTER SET、COLLATE

允许长度：0 到 255（0 到 2^8-1）个字符，另见“其他事项”中的说明。

默认值：如果数据列允许为 NULL，则默认值为 NULL；如果允许为 NOT NULL，默认值为空字符串（''）。

存储空间占用量：数据本身的长度（以字节计算）再加上 1 个用来保存这个长度值的字节。

比较方式：根据数据列所使用的排序方式依次比较各个字符。

其他事项：每个 TINYTEXT 值最多可以由 256 个字节构成。如果其中包含多字节字符，它实际容纳的字符个数将小于 255。

- TEXT [(*M*)]

含义：一个标准尺寸的 TEXT（非二进制字符串）值。

可用属性：BINARY、CHARACTER SET、COLLATE

允许长度：0 到 65535（0 到 $2^{16}-1$）个字符，另见"其他事项"中的说明。如果给出了长度 *M*，*M* 将被用来选择一种适当的数据类型，然后被丢弃。如果 *M* 的值在 1 到 65 535 之间，数据类型变成 TEXT；如果 *M* 的值大于或等于 65 535，数据类型将根据数据值的实际长度变成 MEDIUMTEXT 或 LONGTEXT。

默认值：如果数据列允许为 NULL，则默认值为 NULL；如果允许为 NOT NULL，默认值为空字符串（''）。

存储空间占用量：数据本身的长度（以字节计算）再加上 2 个用来保存这个长度值的字节。

比较方式：根据数据列所使用的排序方式依次比较各个字符。

其他事项：每个 TEXT 值最多可以由 65 535 个字节构成。如果其中包含着多字节字符，它实际容纳的字符个数将小于 65 535。

- MEDIUMTEXT

含义：一个中等尺寸的TEXT（非二进制字符串）值。

可用属性：BINARY、CHARACTER SET、COLLATE

允许长度：0 到 16 777 215（0 到 $2^{24}-1$）个字符，另见"其他事项"中的说明。

默认值：如果数据列允许为 NULL，则默认值为 NULL；如果允许为 NOT NULL，默认值为空字符串（''）。

存储空间占用量：数据本身的长度（以字节计算）再加上 3 个用来保存这个长度值的字节。

比较方式：根据数据列所使用的排序方式依次比较各个字符。

其他事项：每个 MEDIUMTEXT 值最多可以由 16 777 215 个字节构成。如果其中包含着多字节字符，它实际容纳的字符个数将小于 16 777 215。

同义词：LONG VARCHAR

- LONGTEXT

含义：一个大尺寸的TEXT（非二进制字符串）值。

可用属性：BINARY、CHARACTER SET、COLLATE

允许长度：0 到 4 294 967 295（0 到 $2^{32}-1$）个字符；另见"其他事项"中的说明。

默认值：如果数据列允许为 NULL，则默认值为 NULL；如果允许为 NOT NULL，默认值为空字符串（''）。

存储空间占用量：数据本身的长度（以字节计算）再加上 4 个用来保存这个长度值的字节。

比较方式：根据数据列所使用的排序方式依次比较各个字符。

其他事项：每个 LONGTEXT 值最多可以由 4 294 967 295 个字节构成。如果其中包含着多字节字符，它实际容纳的字符个数将小于 4 294 967 295。

B.2.3　ENUM和SET类型

- ENUM（'*value1*', '*value2*', ...）

  含义：枚举值。数据列的取值必须是且仅是合法取值列表中的一个成员。

  可用属性：CHARACTER SET、COLLATE

  默认值：如果数据列允许为 NULL，默认值为 NULL；如果允许为 NOT NULL，则为枚举集合中的第 1 个成员。

  存储空间占用量：如果成员个数在 1 到 255 之间，占用 1 个字节；如果在 256 到 65535 之间，占用 2 个字节。

  比较方式：根据数据列值的数值进行比较。

  其他事项：在数据类型定义里，各成员值里的任何尾缀空格都将被忽略。

- SET（'*value1*', '*value2*', ...）

  含义：集合。数据列的取值可以是任何一种由该集合 0 个或者多个成员构成的子集。

  可用属性：CHARACTER SET、COLLATE

  默认值：如果数据列允许为 NULL，则默认值为 NULL；如果允许为 NOT NULL，默认值为空集合（''）。

  存储空间占用量：1 个字节（1 到 8 个成员的集合）、2 个字节（9 到 16 个成员）、3 个字节（17 到 24 个成员）、4 个字节（25 到 32 个成员）或 8 个字节（33 到 64 个成员）。

  比较方式：根据数据列值的数值进行比较。

  其他事项：在数据类型定义里，各成员值里的任何尾缀空格都将被忽略。

B.3　日期与时间类型

MySQL 提供了各种数据列类型来表示日期/时间值。这些类型可以单独使用，也可以组合使用。有一种特殊的时间戳类型会在数据行修改时自动更新为当前时刻，还有一种表示年的类型则给那些不需使用一个完整日期/时间的人们提供了方便。

在以下内容里，日期格式中的 *CC*、*YY*、*MM*、*DD* 分别代表着世纪、年、月、日。时间格式中的 *hh*、*mm*、*ss* 分别代表着小时、分钟、秒。

- DATE

  含义：日期值，格式为 '*CCYY-MM-DD*'。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  表示范围：'1000-01-01' 到 '9999-12-31'

  零值：'0000-00-00 '

  默认值：如果数据列允许使用 NULL 值，则为 NULL；如果带 NOT NULL 属性，则为'0000-00-00'。

  存储空间占用量：3 个字节。

- DATETIME

  含义：日期加时间值，格式为 '*CCYY-MM-DD hh*:*mm*:*ss*'。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  表示范围：'1000-01-01 00:00:00' 到 '9999-12-31 23:59:59'

  零值：'0000-00-00 00:00:00'

默认值：如果数据列允许使用 NULL 值，则为 NULL；如果带 NOT NULL 属性，则为零值。
存储空间占用量：8 个字节。

- TIME
  含义：时间值，格式为'*hh*:*mm*:*ss*'（负值被表示为'-*hh*:*mm*:*ss*'格式）。
  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。
  表示范围：'-838:59:59'到'838:59:59'
  零值：'00:00:00'
  默认值：如果数据列允许使用 NULL 值，则为 NULL；如果带 NOT NULL 属性，则为零值。
  存储空间占用量：3 个字节。
  其他事项：当你试图把一个非法值插入一个 TIME 数据列时，MySQL 实际填入的将是零值'00:00:00'。但这个零值同时也是正常表示范围内的一个合法值。
- TIMESTAMP
  含义：时间戳（日期+时间）值，格式为'*CCYY-MM-DD hh*:*mm*:*ss*'。TIMESTAMP 类型有几个特点。首先，把 NULL 值插入数据表任何一个 TIMESTAMP 数据列的做法都将当时的日期和时间记录到这个数据列里去，除非数据列允许为 NULL 值。其次，每个数据表的每个 TIMESTAMP 列可以有两个自动更新属性：自动初始化，当创建了一个新行时，这列的默认值为当前时间和日期；自动更新，当某行更新时，修改此行任何其他列的值都将使 TIMESTAMP 列的日期和时间更新为发生修改时的值。你可指定采用此方式的 TIMESTAMP，也可禁止自动初始化、更新等。参见第 3 章。
  可用属性：在同一个数据表里，一个 TIMESTAMP 数据列可以具备 DEFAULT CURRENT_TIMASTAMP 或 ON UPDATE CURRENT_TIMASTAMP 属性，或是同时具备这两种属性。（既不允许把它们分别用于两个不同的 TIMESTAMP 数据列，也不允许把它们当中的任何一个用于一个以上的 TIMESTAMP 数据列。）在创建一个新数据行的时候，带有 DEFAULT CURRENT_TIMASTAMP 属性的 TIMASTAMP 数据列将被设置为当前日期和时间——如果你没有为该数据列提供值的话。在同一个数据行里的其他数据列的值发生变化的时候，带有 ON UPDATE CURRENT_TIMASTAMP 属性的 TIMASTAMP 数据列将被更新为当前的日期和时间。CURRENT_TIMASTAMP()和 NOW()函数相当于 CURRENT_TIMASTAMP 属性的同义词。
  可以在 DEFAULT 关键字的后面给出一个常数值，把某个 TIMASTAMP 数据列设置为一个特定的日期和时间值或零值。
  TIMASTAMP 数据列还可以有 NULL 属性，这将允许它存储 NULL 值。如果没有这种属性，把一个 NULL 值存入该数据列的实际效果将是把它设置为当前的日期和时间。
  取值范围：UTC（Universal Coordinated Time，国际协调时间）'1970-01-01 00:00:01' 到 '2038-01-19 03:14:07'
  零值：'0000-00-00 00:00:00'
  默认值：如果是带 DEFAULT CURRENT_TIMASTAMP 属性的 TIMASTAMP 数据列，DESCRIBE 和 SHOW COLUMNS 语句将把它的默认值显示为 CURRENT_TIMASTAMP；否则，显示常数型的日期/时间默认值。请参阅“可用属性”部分的讨论内容。
  存储空间占用量：4 个字节。
  其他事项：在创建数据表的时候，TIMASTAMP 数据列还会受到 SQL 模式设置的影响。如果启

用了 MAXDBSQL 模式，MySQL 将把 TIMASTAMP 数据列创建为一个 DATATIME 数据列，而这是为了和 MaxDB DBMS 保持兼容。

- ❑ YEAR [(*M*)]

  含义：一个年份值。如果给出的话，*M* 只能是 2（以两位数 *YY* 表示年份） 或者 4（以四位数 *CCYY* 表示年份）；如果省略，*M* 的默认值是 4。

  可用属性：除本附录开头部分介绍的全局属性外，没有其他特殊属性。

  表示范围：YEAR(4)类型为 1901 年到 2155 年以及 0000 年；YEAR(2)类型为 1970 年到 2069 年。但只显示最后两位数字。

  零值：YEAR(4)类型的零值是 0000，YEAR(2)类型的零值是 00。

  默认值：如果数据列允许使用 NULL 值，则为 NULL；如果带 NOT NULL 属性，则为零值。

  存储空间占用量：1 个字节。

B.4 空间类型

这些类型用来表示坐标或几何值。在 MySQL 里，空间值可以表示为 Well-Known Text、Well-Known Binary 或 MySQL 内部专用坐标格式。这里描述的空间数据类型用来保存 MySQL 内部专用格式的坐标值。

只有在被编译时增加了空间数据类型支持功能的 MySQL 服务器才允许使用坐标数据，这由系统变量 have_geometry 的值指定。

对空间类型以及为检索坐标数据而创建的索引类型的支持随存储引擎的不同而变化。MyISAM 数据表既支持空间类型，也支持为检索坐标数据而创建的 SPATIAL 和非 SPATIAL 索引。其他存储引擎如 InnoDB 和 ARCHIVE 虽然支持空间类型，但只允许非 SPATIAL 索引。关于这方面的更多信息见 3.2.7 节。

对于所有的空间类型，允许使用的属性只有 NULL 和 NOT NULL。

- ❑ GEOMETRY

  含义：一个几何对象。这种类型可以容纳任意空间类型的一个值。

- ❑ GEOMETRYCOLLECTION

  含义：由一个或多个几何对象构成的集合，其成员的值可以是任何一种空间类型。

- ❑ LINESTRING

  含义：一条曲线，表示为由一个或多个POINT值构成的集合。

- ❑ MULTILINESTRING

  含义：由一个或多个LINESTRING值构成的集合。

- ❑ MULTIPOINT

  含义：由一个或多个POINT值构成的集合。

- ❑ MULTIPOLYGON

  含义：由一个或多个POLYGON值构成的集合。

- ❑ POINT

  含义：一个点（一组 *X* / *Y* 坐标）。

- ❑ POLYGON

  含义：一个多边形，表示为由一个或多个简单的、封闭的LINESTRING值构成的集合。

附录 C

操作符与函数用法指南

本附录将介绍可在 SQL 语句中用来构造表达式的操作符和函数。SQL 语句中的表达式就是通过它们构造出来的。各个操作符和函数从 MySQL 5.0 版开始发生的变化将在本附录中指出。

本附录中的操作符和函数用法示例将按以下形式写出：

```
expr                                    → result
```

expr 用来演示如何使用操作符或函数，result 是该表达式的求值结果。例如，下面这行文字的意思是：函数调用 RIGHT('my cat', 3)将产生一个字符串结果'cat'。

```
RIGHT('my cat',3)                       → 'cat'
```

本附录中的示例都可以在 mysql 客户程序里试用和检验。具体做法是：启动 mysql 客户程序，然后依次敲入关键字 SELECT、示例表达式和一个分号，最后再按下 Enter 键。比如下面这样：

```
mysql> SELECT RIGHT('my cat',3);
+-------------------+
| RIGHT('my cat',3) |
+-------------------+
| cat               |
+-------------------+
```

MySQL 不要求 SELECT 语句必须有一个 FROM 子句。这为我们这些希望通过输入各种表达式来熟悉操作符和函数使用方法的人提供了方便。

本附录中的示例给出了完整的 SELECT 语句，如果不这样做，就很难演示那些函数的用法。参见 C.2.6 节。

函数名和单词形式的操作符（比如 BETWEEN）允许以任意的字母大小写形式给出。

本附录还使用了以下几种符号来代表常用的函数参数类型。

- expr。代表一个表达式。根据上下文，它可以是一个数值表达式、一个字符串表达式或者一个日期/时间表达式；它还可以是一个常数、某数据列的名字或者其他表达式。
- str。代表一个字符串。它可以是一个文字字符串、某字符串数据列的名字或者一个求值结果为字符串的表达式。
- n。代表一个整数（必要时，在字母表里与n邻近的字母也将用来代表整数）。
- x。代表一个浮点数（必要时，在字母表里与x邻近的字母也将用来代表浮点数）。

至于那些不怎么常用的参数，我们将在讨论过程中随时定义。操作符或函数调用中的可选项将被安排在方括号（[]）里出现。表达式的求值过程往往会牵涉到有关值的类型转换问题。3.5.2 节对类型

转换的上下文环境和 MySQL 进行类型的规则做了详细的讨论。

C.1　操作符

数据项必须通过操作符才能连接在一起构成表达式，用来完成算术运算、数据比较、二进制位操作和逻辑操作、模式匹配等操作。

C.1.1　操作符的优先级

操作符有着各种各样的优先级。我们把操作符按优先级从高到低的顺序依次列在下面这份清单里。在这份清单里，同一行上的操作符都有着相同的优先级。优先级相同的操作符将按从左至右的顺序依次得到求值，而优先级较高的操作符将在优先级较低的操作符之前求值。

```
BINARY  COLLATE
!
- (unary minus)  ~ (unary bit negation)
^
*  /  DIV  %  MOD
+  -
<<  >>
&
|
<  <=  =  <=>  <>  !=  >=  >  IN  IS  LIKE  REGEXP  RLIKE
BETWEEN  CASE  WHEN  THEN  ELSE
NOT
AND  &&
XOR
OR  ||
:=
```

一元操作符（一元减、一元位求反、NOT、BINARY 和 COLLATE）和操作数的绑定关系比二元操作符更紧密。更准确地说，它们和表达式里紧随其后的那个项结合在一起，而不是和整个表达式的其余部分结合在一起。

```
-2+3                                                  → 1
-(2+3)                                                → -5
```

有几个操作符的优先级取决于 MySQL 服务器的 SQL 模式或 MySQL 软件的版本。

- 如果启用了PIPES_AS_CONCATSQL模式，“||”将变成一个字符串合并操作符，而不是一个逻辑OR操作符，它的优先级将上升到“^”操作符之上、一元操作符之下。
- NOT操作符的优先级比“!”操作符低。如果想让NOT操作符的优先级和“!”操作符一样（这是MySQL 5.0.2版之前的行为），请启用HIGH_NOT_PRECEDENCESQL模式。

C.1.2　归组操作符

这些操作符用来对表达式里的表达项归组以控制运算顺序，还可以用来把多个值归为一条记录。

- `(...)`

  括号可以用来对表达式里的“零件”进行归组。它们可以重写默认的操作符优先级，从而改变表达式中表达项的运算顺序。（请参阅 C.1.1 节。）括号还可以改善表达式的清晰度，让表达

式更容易看明白。使用了多层括号的表达式由内到外求值。

```
1 + 2 * 3 / 4                                    → 2.5000
(((1 + 2) * 3) / 4)                              → 2.2500
```

- (*expr*[, *expr*] …)

  ROW(*expr*[, *expr*] …)

  这些行构造符可以用来表达对两条记录（两组值）进行比较。参加比较的两条记录必须包含个数相同的值。上面给出的两种语法（有或没有 ROW 关键字）是等效的。比如说，如果某个子查询返回的一条记录包含 3 个值，你可以任选以下结构之一，把该条记录与一条包含 3 个值的给定记录进行比较：

```
SELECT ... FROM t2 WHERE (0,1,2) = (SELECT col1, col2, col3 FROM ...);
SELECT ... FROM t2 WHERE ROW(0,1,2) = (SELECT col1, col2, col3 FROM ...);
```

  这些行构造符同样可以用在不涉及子查询的上下文里。下面这条语句是合法的：

```
SELECT * FROM president WHERE (first_name,last_name) = ('John','Adams');
```

C.1.3 算术操作符

这些操作符用来完成各种标准的算术运算。算术操作符的操作对象是数值，不是字符串（MySQL 会自动地把看起来像数字的字符串转换为相应的数值）。

算术操作符遵守以下规则。

- 在数值上下文里出现的字符串将被转换为双精度数。
- 如果两个操作数都是整数，加法（+）、减法（-）和乘法（*）运算将使用64位整数进行。这意味着涉及大整数的算术运算有可能超出64位整数的表示范围而导致不可预料的结果：

```
999999999999999999 * 999999999999999999         → -7527149226598858751
99999999999 * 99999999999 * 99999999999         → -1504485813132150785
18014398509481984 * 18014398509481984           → 0
```

- 如果两个操作数都是整数且至少其中之一带有正负号，计算结果将带有正负号。
- 如果两个操作数至少有一个是浮点数，加法、减法、乘法、除法和求余运算结果的精度将由精度最大的那个操作数决定。
- 在除法运算的结果将被当做整数来使用的上下文里，使用“/”操作符进行的除法运算将使用64位整数进行。
- 在使用“/”操作符对两个精确值进行除法运算的时候，运算结果的小数位数等于被除数的小数位数加上div_precision_increment系统变量的值，该变量的默认值是4。
- 涉及NULL值的算术运算的结果仍将是一个NULL值。

  下面是 MySQL 支持的算术操作符。

- +

  加法操作符。求值结果为两个操作数的和。

```
2 + 2                                            → 4
3.2 + 4.7                                        → 7.9
'43bc' + '21d'                                   → 64
'abc' + 'def'                                    → 0
```

请注意，最后一个示例表明 MySQL 不像某些程序设计语言那样把“+”也用作字符串拼接操作符。字符串在参加算术运算之前会先被转换为数值。那些看起来不像是数值的字符串都将被强行转换为 0。如果想拼接字符串，就应该使用 CONCAT()函数。

- –

  如果出现在两个表达式之间，就是减法操作符，求值结果为这两个操作数的差；如果出现在单个表达式的前面，就是单元求负操作符，求值结果为操作数的负值（也就是翻转操作数的正负符号）。

```
10 - 7                                        → 3
-(10 - 7)                                     → -3
```

- *

  乘法操作符。求值结果为两个操作数的积。

```
2 * 3                                         → 6
2.3 * -4.5                                    → -10.35
```

- /

  除法操作符。求值结果为两个操作数的商。若除数为零，则结果为 NULL。

```
3 / 1                                         → 3.0000
1 / 3                                         → 0.3333
1 / 0                                         → NULL
```

- DIV

  整除。求值结果为两个操作数的商，无小数部分。若除数为零，则结果为 NULL。

```
3 DIV 1                                       → 3
1 DIV 3                                       → 0
1 DIV 0                                       → NULL
```

- %，MOD

  求余操作符。求值结果为整数 m 除以整数 n 的余数。m % n 和 m MOD n 与 MOD(m, n)是一回事。类似于除法的情况，若除数为 0，则结果为 NULL。

```
12 % 4                                        → 0
12 % 5                                        → 2
12 % 0                                        → NULL
```

  如果两个操作数是分数，求模运算产生的是除法后的余数。

```
14.4 % 3.2                                    → 1.6
```

C.1.4　比较操作符

如果比较操作的结果为真，比较操作符的返回值就将是 1；如果比较操作的结果为假，比较操作符的返回值就将是 0。可以对数值或字符串进行比较，并会根据以下原则对操作数进行相应的类型转换。

- 除<=>操作符以外，所有涉及NULL值的比较操作都将被求值为NULL。（“<=>”与“=”功能相当，但NULL=NULL的结果为NULL。而表达式“NULL <=> NULL”将被求值为真。）
- 如果两个操作数都是字符串值，它们之间的比较操作将按该类型的方式进行。对于二进制字

符串，比较操作将一个字节一个字节地比较它们各个字节的数值；对于非二进制字符串，比较操作将一个字符一个字符地按照它们在有关字符集里的排位次序进行比较。如果两个字符串使用的字符集不同，比较操作可能出错或不会求值出有意义的结果。如果两个操作数一个是非二进制字符串，另一个是二进制字符串，它们之间的比较操作就将按二进制字符串方式进行。

- 如果两个操作数都是整数，它们之间的比较操作将按整数数值方式进行。
- 只要十六进制常数不是与数值进行比较，就将被视为二进制字符串比较。
- 除IN()外，如果比较操作的两个操作数一个是TIMESTAMP或DATETIME值，另一个是一个常数，比较操作就会把它们都看做是TIMESTAMP值。这是为了使比较操作能够与各种ODBC应用程序有更好的配合。
- 如果以上规则都不适用，比较操作中的操作数就都将被视为双精度浮点数。注意，字符串与数值之间的比较也属于这种情况。字符串将被转换为一个双精度数值，如果字符串看起来不像一个数字，转换结果就将是0。比如说，字符串'14.3'将被转换为浮点数14.3，但字符串'L4.3'却会被转换为数值0。

下面的例子可以帮助我们进一步理解上面这些规则：

```
2 < 12                                                     → 1
'2' < '12'                                                 → 0
'2' < 12                                                   → 1
```

在第一个比较操作里，两个操作数都是整数，所以它们按数值方式进行比较。在第二个比较操作里，两个操作数都是字符串，所以它们按字符串方式进行比较。在第三个比较操作里，两个操作数一个是整数，另一个是字符串，所以它们将被当做双精度浮点数进行比较。

MySQL 按以下规则对字符串进行比较：二进制字符串之间的比较操作逐字节进行，实际比较的是每个字节的数值大小。非二进制字符串之间的比较操作逐字符进行，实际比较的是每个字符在当前字符集的排序方式下的序列。如果字符串所使用的字符集不同，比较操作将报告出错或无法得到有意义的结果。二进制字符串和非二进制字符串之间的比较操作按二进制字符串比较操作进行。

- =

  如果两个操作数相等，该操作符将返回1；否则，返回0。

  ```
  1 = 1                                                  → 1
  1 = 2                                                  → 0
  'abc' = 'abc'                                          → 1
  'abc' = 'ABC'                                          → 1
  'abc' = 'def'                                          → 0
  'abc' = 0                                              → 1
  ```

  'abc'既等于'abc'又等于'ABC'的原因，是非二进制字符串的字符串比较操作不区分字母大小写。'abc'等于 0 的原因，是 MySQL 会先按规则把它转换为数值再进行比较。由于'abc'看起来不像数值，所以它在和数值进行比较之前会先被转换为 0。

  在非二进制字符串比较操作里，彼此相似但在大小写、重音或其他注音符号上有所区别的字符的“大小”，由操作数的字符集和排序方式决定。

  字符串比较操作通常不区分字母的大小写，但如果涉及二进制字符串，或是非二进制字符串使用了二进制排序方式或区分大小写的排序方式，则另当别论。比如说，如果给出了 BINARY

关键字，或者你正在对 BINARY、VARBINARY 或 BLOB 数据列里的值进行比较，该比较操作将区分字母的大小写：

```
'abc' = 'ABC'                                    → 1
BINARY 'abc' = 'ABC'                             → 0
BINARY 'abc' = 'abc'                             → 1
_latin1 'abc' COLLATE latin1_bin = 'ABC'         → 0
_latin1 'abc' COLLATE latin1_general_cs = 'ABC'  → 0
```

尾缀空格在二进制字符串比较操作里将参加比较，在非二进制字符串比较操作里不参加。

```
BINARY 'a' = 'a '                                → 0
'a' = 'a '                                       → 1
```

- <=>

  等于操作符，允许操作数为 NULL 值。它与“=”操作符相似，不同的是，只要两个操作数相等——即使它们是 NULL 值，求值结果就为 1。

```
1 <=> 1                                          → 1
1 <=> 2                                          → 0
NULL <=> NULL                                    → 1
NULL = NULL                                      → NULL
```

  最后两个示例演示了“<=>”与“=”在操作数为 NULL 值时的区别。

- <>、!=

  如果两个操作数不相等，则求值结果为 1；否则，求值结果为 0。

```
3.4 != 3.4                                       → 0
'abc' <> 'ABC'                                   → 0
BINARY 'abc' <> 'ABC'                            → 1
'abc' != 'def'                                   → 1
```

- <

  如果左操作数小于右操作数，则求值结果为 1；否则，求值结果为 0。

```
3 < 10                                           → 1
105.4 < 10e+1                                    → 0
'abc' < 'ABC'                                    → 0
'abc' < 'def'                                    → 1
```

- <=

  如果左操作数小于或等于右操作数，则求值结果为 1；否则，求值结果为 0。

```
'abc' <= 'a'                                     → 0
'a' <= 'abc'                                     → 1
13.5 <= 14                                       → 1
(3 * 4) - (6 * 2) <= 0                           → 1
```

- >

  如果左操作数大于右操作数，则求值结果为 1；否则，求值结果为 0。

```
PI() > 3                                         → 1
'abc' > 'a'                                      → 1
SIN(0) > COS(0)                                  → 0
```

- >=

  如果左操作数大于或等于右操作数，则求值结果为 1；否则，求值结果为 0。

```
'abc' >= 'a'                              → 1
'a' >= 'abc'                              → 0
13.5 >= 14                                → 0
(3 * 4) - (6 * 2) >= 0                    → 1
```

- *expr* BETWEEN *min* AND *max*

  expr NOT BETWEEN *min* AND *max*

  如果 expr 落在从 min 到 max 的区间内（包括 min 和 max 在内），则 BETWEEN 操作符的求值结果为 1；否则，求值结果为 0。NOT BETWEEN 操作符的求值情况正好与此相反。如果操作数 expr、min 和 max 都是同一种类型，则下面两个表达式等价：

```
expr BETWEEN min AND max
(min <= expr AND expr <= max)
```

  如果这些操作数不是同一类型，就会发生类型转换，上面这两个表达式就不一定等价了。此时，BETWEEN 操作符的求值结果将由 expr 的类型所决定的比较操作来确定：

- 如果expr是一个字符串，这些操作数就将被作为字符串并按本节开头的规则进行比较；
- 如果expr是一个整数，这些操作数就将被作为整数按数值方式进行比较。
- 如果以上两条规则都不适用，这些操作数就将被作为浮点数按数值方式进行比较。

```
'def' BETWEEN 'abc' AND 'ghi'             → 1
'def' BETWEEN 'abc' AND 'def'             → 1
13.3 BETWEEN 10 AND 20                    → 1
13.3 BETWEEN 10 AND 13                    → 0
2 BETWEEN 2 AND 2                         → 1
'B' BETWEEN 'A' AND 'a'                   → 0
BINARY 'B' BETWEEN 'A' AND 'a'            → 1
```

  对于使用混合时间类型或使用混合时间类型加字符串的 BETWEEN 表达式，最好使用 CAST() 确保所有操作数的类型都相同。

- CASE [*expr*] WHEN *expr1* THEN *result1* ...[ELSE *default*] END

  如果存在第一个表达式 expr，CASE 就会把它与每一个 WHEN 后面的表达式进行比较，对于第一个相等的表达式，相应的 THEN 后面的值就是求值结果。这特别适用于需要把一个给定值与一组值进行比较的场合：

```
CASE 0 WHEN 1 THEN 'T' WHEN 0 THEN 'F' END          → 'F'
CASE 'F' WHEN 'T' THEN 1 WHEN 'F' THEN 0 END        → 0
```

  如果没有表达式 expr，CASE 操作符将计算 WHEN 表达式。对于第一个结果为真（既不能是 0，也不能是 NULL）的表达式，相应的 THEN 后面的值就是求值结果。这特别适用于需要判断“不等于”关系或者需要对一系列条件进行测试的场合：

```
CASE WHEN 1=0 THEN 'absurd' WHEN 1=1 THEN 'obvious' END
                                                    → 'obvious'
```

  如果 WHEN 后面的表达式没有一个成立，求值结果就将是 ELSE 后面的值。如果 ELSE 子句不存在，CASE 操作符的求值结果就将是 NULL。

```
CASE 0 WHEN 1 THEN 'true' ELSE 'false' END        → 'false'
CASE 0 WHEN 1 THEN 'true' END                     → NULL
CASE WHEN 1=0 THEN 'true' ELSE 'false' END        → 'false'
CASE WHEN 1/0 THEN 'true' END                     → NULL
```

默认的返回值的聚合类型决定着整个 CASE 表达式的返回类型。

```
CASE 1 WHEN 0 THEN 0 ELSE 1 END                   → 1
CASE 1 WHEN 0 THEN '0' ELSE '1' END               → '1'
```

但是默认返回类型也受上下文影响，上下文可能会引起类型转换，转换为字符串，数字等。注意 CASE 表达式与 E.2.1 节中的 CASE 语句有所不同。

- *expr* IN (*value1*, *value2*, ...)

  expr NOT IN (*value1*, *value2*, ...)

  如果 expr 与列表中的某个值相等，IN()的求值结果就将为 1；否则，就将为 0。“NOT IN()”的求值情况正好与此相反。下面两个表达式是等价的：

  ```
  expr NOT IN (value1, value2,...)
  NOT (expr IN (value1, value2,...))
  ```

  如果列表里的值全都是常数，MySQL 就会对它们进行排序，并利用二元搜索树算法来对 IN()测试进行求值，这个算法是非常快的。

  ```
  3 IN (1,2,3,4,5)                                  → 1
  'd' IN ('a','b','c','d','e')                      → 1
  'f' IN ('a','b','c','d','e')                      → 0
  3 NOT IN (1,2,3,4,5)                              → 0
  'd' NOT IN ('a','b','c','d','e')                  → 0
  'f' NOT IN ('a','b','c','d','e')                  → 1
  ```

- *expr* IS {FALSE | TRUE | UNKNOWN}

  这些语句将 expr 与逻辑 FALSE、TRUE 和 UNKNOWN 比较，返回 0（假）或 1（真）。U 值表示假，非 0 和非 NULL 表示真，NULL 表示未知。

  ```
  2 IS FALSE                                        → 0
  2 IS TRUE                                         → 1
  2 IS UNKNOWN                                      → 0
  NULL IS FALSE                                     → 0
  NULL IS TRUE                                      → 0
  NULL IS UNKNOWN                                   → 1
  ```

- *expr* IS NULL

  expr IS NOT NULL

  如果 expr 的值是 NULL，“IS NULL”的求值结果就将为 1；否则，就将为 0。“IS NOT NULL”的求值情况正好与此相反。下面两个表达式是等价的：

  ```
  expr IS NOT NULL
  NOT (expr IS NULL)
  ```

  “IS NULL”和“IS NOT NULL”专门用来判断 expr 的值是否为 NULL 值。普通的比较操作符“=”、“<>”和“!=”无法进行这种判断。(但可以用操作符“<=>”来测试 NULL 值。)

  ```
  NULL IS NULL                                      → 1
  0 IS NULL                                         → 0
  ```

```
NULL IS NOT NULL                              → 0
0 IS NOT NULL                                 → 1
NOT (0 IS NULL)                               → 1
NOT (NULL IS NULL)                            → 0
```

C.1.5 位操作符

本节介绍用来完成各种位操作的操作符。位操作必须用 BIGINT 值（64 位整数）完成，这就限制了这类操作的最大范围。位操作的结果是 64 位无符号数。如果操作数里有 NULL 值，则位操作的结果就将是 NULL。

- &

  求值结果为两个操作数进行 AND（与）操作得到的结果。

```
1 & 1                                         → 1
1 & 2                                         → 0
7 & 5                                         → 5
```

- |

  求值结果为两个操作数进行 OR（或）操作得到的结果。

```
1 | 1                                         → 1
1 | 2                                         → 3
1 | 2 | 4 | 8                                 → 15
1 | 2 | 4 | 8 | 15                            → 15
```

- ^

  求值结果为两个操作数进行 XOR（异或）操作得到的结果。

```
1 ^ 1                                         → 0
1 ^ 0                                         → 1
255 ^ 127                                     → 128
```

- <<

  把左操作数的各位左移，移动次数由右操作数指定。如果右操作数为负值，则操作结果为 0。

```
1 << 2                                        → 4
2 << 2                                        → 8
1 << 63                                       → 9223372036854775808
1 << 64                                       → 0
```

  最后两个例子演示了 64 位计算的极限情况。

- >>

  把左操作数的各位右移，移动次数由右操作数指定。如果右操作数为负值，则操作结果为 0。

```
16 >> 3                                       → 2
16 >> 4                                       → 1
16 >> 5                                       → 0
```

- ~

  对随后的操作数逐位求反，即把所有的 0 位翻转为 1，把所有的 1 位翻转为 0。

```
~0                                            → 18446744073709551615
~(-1)                                         → 0
~~(-1)                                        → 18446744073709551615
```

C.1.6　逻辑操作符

逻辑操作符也叫做布尔操作符，它们用来测试表达式是否成立（成立为真，不成立为假）。在 MySQL 里，如果逻辑操作符的求值结果为真，则返回 1；如果为假，则返回 0。逻辑操作符把非零、非 NULL 操作数解释为真，把 0 操作数解释为假。逻辑操作符对 NULL 值的处理情况见它们各自的条目。

逻辑操作符要求操作数是数值类型，字符串操作数在求值过程开始之前会被转换为数值。

在 MySQL 里，“！”、“||”和“&&”都是逻辑操作符，就像它们在 C 语言里一样。特别地，“||”不像标准 SQL 语言里规定的那样用来完成字符串合并操作。要进行字符串合并，应使用 CONCAT()函数。如果想把“||”当做字符串合并操作符来使用，需要启用 PIPES_AS_CONCAT SQL 模式。

- NOT 或 !

  逻辑非操作符。如果随后的操作数为假，则求值结果为 1；如果操作数为真，则求值结果为 0。但 NOT NULL 仍将为 NULL。

  ```
  NOT 0                    → 1
  NOT 1                    → 0
  NOT NULL                 → NULL
  NOT 3                    → 0
  NOT NOT 1                → 1
  NOT '1'                  → 0
  NOT '0'                  → 1
  NOT 'abc'                → 1
  ```

  在上面的最后几个例子里，字符串操作数将被转换为数值后再求值。

  NOT 操作符的优先级可以按照 C.1.1 节加以更改。

- AND 或 &&

  逻辑“与”操作符。如果两个操作数都是真（既不能是 0，也不能是 NULL），则求值结果为 1；如果有一个操作数为假，则为 0；否则，求值结果为 NULL（结果不确定）。

  ```
  4 AND 2                  → 1
  0 AND 0                  → 0
  0 AND 3                  → 0
  1 AND NULL               → NULL
  0 AND NULL               → 0
  NULL AND NULL            → NULL
  ```

- OR 或 ||

  逻辑“或”操作符。只要两个操作数里有一个为真（既不能是 0，也不能是 NULL），则求值结果为 1；如果有一个操作数为假，求值结果为 0；否则，为 NULL（结果不确定）。

  ```
  4 OR 2                   → 1
  0 OR 3                   → 1
  0 OR 0                   → 0
  1 OR NULL                → 1
  0 OR NULL                → NULL
  NULL OR NULL             → NULL
  ```

- XOR

  逻辑“异或”操作符。如果有且仅有一个操作数为真（既不能是 0，也不能是 NULL），则求值结果为 1；否则，求值结果为 0。如果操作数中有 NULL 值，求值结果为 NULL。

```
0 XOR 0                                          → 0
0 XOR 9                                          → 1
7 XOR 0                                          → 1
5 XOR 2                                          → 0
```

C1.7　类型转换操作符

类型转换操作符能够把数据值从一个类型转换为另一个类型。

- `_charset str`

  这个操作符用来利用一个给定的字符集解释字符串常数或者某个数据列里的值，charset 必须是服务器所支持的某个字符集的名字。比如说，下面的表达式将分别使用 latin2、utf8 字符集解释字符串'abcd'：

  ```
  _latin2 'abcd'
  _utf8 'abcd'
  ```

  对于多字节字符集的引导符，万一它的操作数尾部字节的个数不够创建一个完整的字符，转换结果的尾部可能会有一些空格。

- `BINARY str`

  BINARY 操作符用来把紧随其后的操作数转换为一个二进制字符串。对结果的比较将使用每个类型的数值逐个字节进行。如果紧随其后的操作数是一个数值，就先把它转换为字符串形式：

  ```
  'abc' = 'ABC'                                → 1
  'abc' = BINARY 'ABC'                         → 0
  BINARY 'abc' = 'ABC'                         → 0
  '2' < 12                                     → 1
  '2' < BINARY 12                              → 0
  ```

  在最后一个例子里，BINARY 操作符强制进行一次由数值到字符串的转换。然后，因为两个操作数都是字符串，所以比较操作将按二进制字符串方式进行。

- `str COLLATE collation`

  COLLATE 操作符将使给定字符串 str 按字符集 str 的排位次序被比较。这对比较操作、排序操作、归组操作以及 DISTINCT 等操作都会产生影响，如下所示：

  ```
  SELECT ... WHERE utf8_str COLLATE utf8_icelandic_ci > 'M';
  SELECT MAX(greek_str COLLATE greek_general_ci) FROM ... ;
  SELECT ... GROUP BY latin1_str COLLATE latin1_german2_ci;
  SELECT ... ORDER BY sjis_str COLLATE sjis_bin;
  SELECT DISTINCT latin2_str COLLATE latin2_croatian_ci FROM ...;
  ```

C.1.8　模式匹配操作符

MySQL 提供了两种模式匹配机制：一种是使用 LIKE 操作符的 SQL 模式匹配，另一种是使用 REGEXP 操作符的正则表达式模式匹配。只有整个字符串得到匹配时，SQL 模式才算匹配成功。而只要能在字符串里找到匹配模式，正则表达式模式就算匹配成功。

- *str* LIKE *pattern* [ESCAPE 'c']

  str NOT LIKE *pattern* [ESCAPE 'c']

  LIKE 是 SQL 模式匹配操作的操作符，当匹配模式 pat 与字符串表达式 str 完整地匹配时，

它的求值结果将是 1。如果没有得到匹配，LIKE 操作符的求值结果将是 0。NOT LIKE 操作符的求值情况则正好相反。下面两个表达式是等价的：

```
str NOT LIKE pattern [ESCAPE 'c']
NOT (str LIKE pattern [ESCAPE 'c'])
```

只要两个操作数里有一个是 NULL，求值结果就将是 NULL。

在 SQL 模式里，有两个字符是有着特殊含义的通配符。

- "%"——能与除 NULL 以外的任意字符序列（包括空字符序列）相匹配。
- "_"——能与任何一个字符相匹配。

SQL 模式允许混合使用这两种通配符，如下所示：

```
'catnip' LIKE 'cat%'                                    → 1
'dogwood' LIKE '%wood'                                  → 1
'bird' LIKE '____'                                      → 1
'bird' LIKE '___'                                       → 0
'dogwood' LIKE '%wo__'                                  → 1
```

如果有一个操作数是二进制字符串，那么 LIKE 就会按二进制字符串比较字符串。如果操作数是非二进制字符串，就使用操作数排序方式比较。

```
'abc' LIKE 'ABC'                                        → 1
BINARY 'abc' LIKE 'ABC'                                 → 0
'abc' LIKE BINARY 'ABC'                                 → 0
'abc' LIKE 'ABC' COLLATE latin1_general_ci              → 1
'abc' LIKE 'ABC' COLLATE latin1_general_cs              → 0
```

因为通配符"%"能够与任何一个字符序列相匹配，所以它甚至能与空字符串相匹配：

```
'' LIKE '%'                                             → 1
'cat' LIKE 'cat%'                                       → 1
```

MySQL 允许使用 LIKE 操作符对数值表达式进行匹配：

```
50 + 50 LIKE '1%'                                       → 1
200 LIKE '2__'                                          → 1
```

如果想对通配符进行匹配，就必须给它们加上一个前导的反斜线字符"\"以取消其特殊含义，如下所示：

```
'100% pure' LIKE '100%'                                 → 1
'100% pure' LIKE '100\%'                                → 0
'100% pure' LIKE '100\% pure'                           → 1
```

要解释"\"，应启用 NO_BACKSLASH_ESCAPES SQL 模式。或者，重新定义转义字符，指定一个 ESCAPE 子句。

```
'100% pure' LIKE '100^%' ESCAPE '^'                     → 0
'100% pure' LIKE '100^% pure' ESCAPE '^'                → 1
```

- *str* REGEXP *pattern*

  str NOT REGEXP *pattern*

  REGEXP 是正则表达式模式匹配操作的操作符，只要能在字符串表达式 str 里找到匹配模式字符串 pattern，它的求值结果将是 1；否则，它的求值结果就将是 0。NOT REGEXP 操作符的求值情况正好与 REGEXP 操作符相反。下面两个表达式是等价的：

```
str NOT REGEXP pattern
NOT (str REGEXP pattern)
```

如果有一个操作数为NULL，正则表达式的匹配结果将为NULL。

如果两个操作数当中至少有一个是二进制字符串，REGEXP操作符将把字符串当做二进制字符串来比较；如果两个操作数都是非二进制字符串，REGEXP 操作符将根据操作数的排序方式比较。

```
'abc' REGEXP 'ABC'                          → 1
BINARY 'abc' REGEXP 'ABC'                   → 0
'abc' REGEXP BINARY 'ABC'                   → 0
'abc' REGEXP 'ABC' COLLATE latin1_bin       → 0
'abc' COLLATE latin1_bin REGEXP 'ABC'       → 0
```

REGEXP操作符目前尚不支持多字节字符集，只适用于单字节字符集。

正则表达式与Unix实用工具程序grep和sed使用的模式相似（如表C-1所示）。

表 C-1

元　素	含　义
^	匹配字符串的开头
$	匹配字符串的结尾
.	匹配任何一个单个的字符，包括换行符
[...]	匹配方括号内的任何一个字符
[^ ...]	匹配没有出现在方括号内的任何一个字符
*e**	匹配模式元素e的0次或更多次出现
e+	匹配模式元素e的1次或更多次出现
e?	匹配模板元素e的0次或1次出现
e1 | *e2*	匹配模式元素e1或e2
e{*m*}	匹配模式元素e的m次出现
e{*m*, }	匹配模式元素e的*m*次或更多次出现
e{, *n*}	匹配模式元素e的0次到*n*次出现
e{*m*, *n*}	匹配模式元素e的*m*次到*n*次出现
(...)	把括号中的模式元素当做一个元素来对待
其他	非特殊字符将与自身匹配

正则表达式模式不要求匹配模式与整个字符串相匹配，只要能在字符串里找到匹配模式就足够了，如下所示：

```
'cats and dogs' REGEXP 'dogs'               → 1
'cats and dogs' REGEXP 'cats'               → 1
'cats and dogs' REGEXP 'c.*a.*d'            → 1
'cats and dogs' REGEXP 'o'                  → 1
'cats and dogs' REGEXP 'x'                  → 0
```

“^”和“$”分别用来匹配字符串的开头和末尾：

```
'abcde' REGEXP 'b'                          → 1
'abcde' REGEXP '^b'                         → 0
'abcde' REGEXP 'b$'                         → 0
```

```
'abcde' REGEXP '^a'                        → 1
'abcde' REGEXP 'e$'                        → 1
'abcde' REGEXP '^a.*e$'                    → 1
```

“[...]”或“[^...]”构造用来设定字符分组。在字符分组里，可以用连字符（-）来设定一个字符区间，连字符的两端是该字符区间的起始字符和结尾字符。比如说，[a-c]将匹配任何一个小写字符，而[0-9]将匹配任何一个数字：

```
'bin' REGEXP '^b[aeiou]n$'                 → 1
'bxn' REGEXP '^b[aeiou]n$'                 → 0
'oboeist' REGEXP '^ob[aeiou]+st$'          → 1
'wolf359' REGEXP '[a-z]+[0-9]+'            → 1
'wolf359' REGEXP '[0-9a-z]+'               → 1
'wolf359' REGEXP '[0-9]+[a-z]+'            → 0
```

如果想把字符“]”放到一个字符分组里，就必须把它放在该字符分组的第一个。如果想把字符“-”放到一个字符分组里，就必须把它放在该字符分组的第一个或者最后一个。如果想把字符“^”放到一个字符分组里，它就不能是“[”后面的第一个字符。

POSIX 标准定义了一些字符分组（character class）以方便人们在构造正则表达式时使用。表 C-2 列出了一些在 MySQL 里比较常用的字符分组，其中有几个等价于上面提到的字符区间。需要特别注意的是，POSIX 字符分组的名字里包含的“[”和“]”方括号字符不可遗漏，所以你在使用字符分组构造正则表达式的时候千万要写出数量足够的方括号。

表　C-2

元　　素	含　　义
[:alnum:]	字母和数字字符
[:alpha:]	字母字符
[:blank:]	空白字符（空格或制表符）
[:cntrl:]	控制字符
[:digit:]	十进制数字（0~9）
[:graph:]	图形字符（不包括空白字符）
[:lower:]	小写字母字符
[:print:]	图形或空格字符
[:punct:]	标点符号字符
[:space:]	空格、制表符、换行符或回车符
[:upper:]	大写字母字符
[:xdigit:]	十六进制数字（0~9，a~f，A~F）

在使用时，POSIX 构造要放在一个字符分组里：

```
'abc' REGEXP '[[:space:]]'                 → 0
'a c' REGEXP '[[:space:]]'                 → 1
'abc' REGEXP '[[:digit:][:punct:]]'        → 0
'a0c' REGEXP '[[:digit:][:punct:]]'        → 1
'a,c' REGEXP '[[:digit:][:punct:]]'        → 1
```

在字符分组里，特殊标记“[:<:]”和“[:>:]”分别匹配单词边界的开始和结束。alpha 字符分组里的所有字符和下划线字符统称为“单词字符”，而“单词”则是由一个或多个单词字

符构成的字符串，该字符串前面和后面的字符必须是非单词字符。

```
'a few words' REGEXP '[[:<:]]few[[:>:]]'          → 1
'a few words' REGEXP '[[:<:]]fe[[:>:]]'           → 0
```

MySQL还允许在正则表达式字符串里对转义序列使用类似C语言的语法。比如说，“\n”、“\t”、“\\”将分别被解释为换行符、制表符和反斜线字符（\）。当需要在匹配模式里使用这几个字符时，必须双写反斜线字符（即把它们分别写成“\\n”、“\\t”、“\\\\”）。语法分析器在对查询语句进行分析时会去掉一个反斜线字符，模式匹配操作再把剩下来的转义序列解释为相应的字符。

■ *str* RLIKE *pattern*

str NOT RLIKE *pattern*

RLIKE 和 NOT RLIKE 是 REGEXP 和 NOT REGEXP 的同义词。

C.2　函数

函数在调用后会返回一个值。在默认的情况下，调用时函数名与紧随其后的左括号之间不允许出现空格，如下所示：

```
mysql> SELECT NOW();
+---------------------+
| NOW()               |
+---------------------+
| 2008-04-30 22:39:26 |
+---------------------+
mysql> SELECT NOW ();
ERROR 1305 (42000): FUNCTION  NOW does not exist
```

如果启用了 IGNORE_SPACE SQL 模式，函数名与紧随其后的左括号之间就允许有空格，但这种做法的副作用是所有的函数名都将被视为保留字。你也可以在建立服务器连接的时候根据客户程序选择这种行为。例如，在启动 mysql 客户程序时，给它加上--ignore-space 选项；在 C 程序里，调用 mysql_real_connect()函数时给它加上 CLIENT_IGNORE_SPACE 选项。

在大多数场合，你可以用逗号来分隔某个函数的多个输入参数，函数参数的前后也允许出现空格。下面两行都是合法的：

```
CONCAT('abc','def')
CONCAT( 'abc' , 'def' )
```

但有些函数不允许这样做，比如 TRIM()或 EXTRACT()函数：

```
TRIM(' ' FROM 'x')                            → 'x'
EXTRACT(YEAR FROM '2003-01-01')               → 2003
```

我们具体介绍每个函数时将注明它们的语法。

C.2.1　比较类函数

下面这些函数用于比较值。

❑ ELT (*n*, *str1*, *str2*,...)

这个函数的返回值是字符串列表 str1、str2、...里的第 *n* 个字符串。如果 *n* 是 NULL、第 *n*

个字符串是 NULL 或者不存在，则返回 NULL。列表里的第一个字符串的索引是 1。ELT()和 FIELD()函数之间呈互补关系。

```
ELT(3,'a','b','c','d','e')                          → 'c'
ELT(0,'a','b','c','d','e')                          → NULL
ELT(6,'a','b','c','d','e')                          → NULL
ELT(FIELD('b','a','b','c'),'a','b','c')             → 'b'
```

- FIELD (*arg0*, *arg1*, *arg2*,...)

  在参数列表 *arg1*, *arg2*,...里找到与 *arg0* 相匹配的那个参数并返回该参数的索引（从 1 开始）。如果没能找到匹配或者 *arg0* 是 NULL，则返回 0。如果所有的参数都是字符串，进行字符串比较；如果所有的参数都是数值，进行数值比较；其他情况进行双精度浮点比较。FIELD()和 ELT()函数之间呈互补关系。

```
FIELD('b','a','b','c')                              → 2
FIELD('d','a','b','c')                              → 0
FIELD(NULL,'a','b','c')                             → 0
FIELD(ELT(2,'a','b','c'),'a','b','c')               → 2
```

- GREATEST(*expr1*, *expr2*, ...)

  返回值是输入参数中的最大值。这个“最大值”是根据以下原则确定的。

  - 如果这个函数是在整数上下文里被调用的，或者它的输入参数全都是整数，那些输入参数就将按整数方式进行比较。
  - 如果这个函数是在浮点数上下文里被调用的，或者它的输入参数全都是浮点数，那些输入参数就将按浮点数方式进行比较。
  - 如果以上两条规则都不适用，那些输入参数就将按字符串方式进行比较。C.1.4节开头描述了字符串比较操作。

```
GREATEST(2,3,1)                                     → 3
GREATEST(38.5,94.2,-1)                              → 94.2
GREATEST('a','ab','abc')                            → 'abc'
GREATEST(1,3,5)                                     → 5
GREATEST('A','b','C')                               → 'C'
GREATEST(BINARY 'A','b','C')                        → 'b'
```

- IF(*expr1*, *expr2*, *expr3*)

  若表达式 *expr1* 为真（既不能是 0，也不能是 NULL），则返回 *expr2*；否则，返回 *expr3*。如果 *expr2* 或 *expr3* 是字符串，则 IF()返回一个字符串；如果 *expr2* 和 *expr3* 中有一个是浮点值，则 IF()返回一个浮点值；如果 *expr2* 和 *expr3* 中有一个是整数，则返回一个整数。

```
IF(1,'true','false')                                → 'true'
IF(0,'true','false')                                → 'false'
IF(NULL,'true','false')                             → 'false'
IF(1.3,'non-zero','zero')                           → 'non-zero'
IF(0.3 <> 0,'non-zero','zero')                      → 'non-zero'
```

  注意，IF()函数与 E.2.1 节中的 IF 函数不同。

- IFNULL(*expr1*, *expr2*)

  若表达式 *expr1* 的值为真（非 0 或非 NULL），则返回 *expr2*；否则，返回 *expr1*。IFNULL()函数将根据自己被调用时的上下文来决定是返回一个数值还是返回一个字符串。如下所示：

```
IFNULL(NULL,'null')                              → 'null'
IFNULL('not null','null')                        → 'not null'
```

- INTERVAL(*n*, *n1*, *n2*, ...)

  如果 *n*<*n1*，返回 0；如果 *n*<*n2*，返回 1；依此类推。如果 *n* 是 NULL，则返回-1。也就是说，INTERVAL()函数可以让我们了解它的第一个参数位于后续参数所定义的第几个区间（*n1* 和 *n2* 构成第一个区间，*n2* 和 *n3* 构成第二个区间，依此类推）。这个函数的所有参数都必须是整数。根据这个函数所使用的快速二进制搜索算法的要求，参数 *n1*、*n2*…的值必须严格按照递增顺序排列（*n1*< *n2*< ...）；如果不是这样，INTERVAL()函数的行为将难以预料。

```
INTERVAL(2,0,1,3)                                → 2
INTERVAL(7,1,3,5,7,9)                            → 4
```

- ISNULL(*expr*)

  若表达式 *expr* 的值是 NULL，则返回 1；否则，返回 0。

```
ISNULL(NULL)                                     → 1
ISNULL(0)                                        → 0
ISNULL(1)                                        → 0
```

- LEAST(*expr1*, *expr2*, ...)

  返回值是输入参数中的最小值。用来确定“最小值”的原则与 GREATEST()函数的一样。

```
LEAST(2,3,1)                                     → 1
LEAST(38.5,94.2,-1)                              → -1
LEAST('a','ab','abc')                            → 'a'
```

- NULLIF(*expr1*, *expr2*)

  若作为其输入参数的两个表达式的值不同，返回 *expr1*；若它们的值相同，则返回 NULL。

```
NULLIF(3,4)                                      → 3
NULLIF(3,3)                                      → NULL
```

- STRCMP(*str1*, *str2*)

  这个函数将根据字符串 *str1* 是否大于、等于、小于字符串 *str2* 而返回 1、0、-1。只要两个参数中有一个是 NULL 值，这个函数就会返回 NULL。如果两个参数中至少有一个是二进制字符串，STRCMP()就按二进制字符串比较。如果操作数都是非二进制字符串，则按操作数排序方式比较。

```
STRCMP('a','a')                                  → 0
STRCMP('a','A')                                  → 0
STRCMP(BINARY 'a','A')                           → 1
STRCMP('A' COLLATE latin1_general_ci,'a')        → 0
STRCMP('A' COLLATE latin1_general_cs,'a')        → -1
```

C.2.2 类型转换函数

这类函数能够把数据值从一种类型转换为另一种类型。

- CAST(*expr* AS *type*)

  把表达式 expr 的值转换为给定的类型。其中，*type* 可以是 BINARY(*n*)（二进制字符串）、CHAR(*n*)（非二进制字符串）、DATE、DATETIME、TIME、SIGNED [INTEGER]、UNSIGNED [INTEGER]或 DECIMAL[(*M*[,*D*])]（自 MySQL 5.0.8 起）。

```
CAST(304 AS BINARY)                              → '304'
CAST(-1 AS UNSIGNED)                             → 18446744073709551615
CAST(13 AS DECIMAL(5,2))                         → 13.00
```

如果参数 *type* 是 BINARY 或 CHAR，还可以给出一个可选的长度参数 n，把转换结果的长度限制在 *n* 个字节或 *n* 个字符内。从 MySQL 5.0.17 版开始，如果参数 *type* 是 BINARY，少于 *n* 个字节的值将用尾缀的零值字节（0x00）补齐到长度等于 *n* 为止。

在需要使用 CREATE TABLE ... SELECT 语句来创建一个新数据表的时候，人们经常会利用 CAST()函数把某些数据列强行设置为指定的类型。如下所示：

```
mysql> CREATE TABLE t SELECT CAST(20080101 AS DATE) AS date_val;
mysql> SHOW COLUMNS FROM t;
+----------+------+------+-----+---------+-------+
| Field    | Type | Null | Key | Default | Extra |
+----------+------+------+-----+---------+-------+
| date_val | date | YES  |     | NULL    |       |
+----------+------+------+-----+---------+-------+
mysql> SELECT * FROM t;
+------------+
| date_val   |
+------------+
| 2008-01-01 |
+------------+
```

CAST()与 CONVERT()函数比较相似，但前者遵守的是标准 SQL 语法，而后者遵守的却是 ODBC 语法。

- ❑ CONVERT(*expr*,*type*)

  CONVERT(*expr* USING *charset*)

  除使用的语法稍微有些不同外，CONVERT()函数的第一种形式与 CAST()函数的功能完全相同，*expr* 和 *type* 参数的用法也完全一致。第二种（USING）形式用来把值转换到指定的字符集上。

```
CONVERT(304,BINARY)                              → '304'
CONVERT(-1,UNSIGNED)                             → 18446744073709551615
CONVERT('abc' USING utf8);                       → 'abc'
```

C.2.3　数值函数

如果你传递给某个数值类函数的输入参数超出了它的允许范围或者是一个非法值，这个函数就会返回一个 NULL。

- ❑ ABS(*x*)

  返回值：*x* 的绝对值。

```
ABS(13.5)                                        → 13.5
ABS(-13.5)                                       → 13.5
```

- ❑ ACOS(*x*)

  返回值：*x* 的反余弦值。如果 *x* 不在−1 到 1 的区间内，则返回 NULL。

```
ACOS(1)                                          → 0
ACOS(0)                                          → 1.5707963267949
ACOS(-1)                                         → 3.1415926535898
```

- ❑ ASIN(*x*)

  返回值：*x* 的反正弦值。如果 *x* 不在−1 到 1 的区间内，则返回 NULL。

```
ASIN(1)                                → 1.5707963267949
ASIN(0)                                → 0
ASIN(-1)                               → -1.5707963267949
```

- ❑ ATAN(*x*)

  ATAN(*y*, *x*)

  只带一个输入参数的 ATAN() 函数将返回 *x* 的反正切值；带两个输入参数的函数是 ATAN2() 函数的一个同义词。

```
ATAN(1)                                → 0.78539816339745
ATAN(0)                                → 0
ATAN(-1)                               → -0.78539816339745
```

- ❑ ATAN2(*y*, *x*)

  相当于 ATAN2(*y*/*x*)，但它要根据两个输入参数的正负符号来判断返回值将落在哪一个坐标象限里。

```
ATAN2(1,1)                             → 0.78539816339745
ATAN2(1,-1)                            → 2.3561944901923
ATAN2(-1,1)                            → -0.78539816339745
ATAN2(-1,-1)                           → -2.3561944901923
```

- ❑ CEILING(*x*)

  CEIL(*x*)

  返回不小于 *x* 的最小整数。如果参数 *x* 是某种精确数值类型的，返回值也将有着同样的类型；否则，返回值将有着某种浮点（近似）数值类型，即使这个函数的返回值没有小数部分。

```
CEILING(3.8)                           → 4
CEILING(-3.8)                          → -3
```

- ❑ COS(*x*)

  返回值：*x* 的余弦值。*x* 被视为是一个弧度值。

```
COS(0)                                 → 1
COS(PI())                              → -1
```

- ❑ COT(*x*)

  返回值：*x* 的余切值。*x* 被视为是一个弧度值。

```
COT(PI()/4)                            → 1
```

- ❑ CRC32(str)

  返回值为字符串 str 的循环冗余校验值，这个返回值是一个 32 位（即 0 到 $2^{32}-1$ 之间）的无符号整数值。如果输入参数是 NULL 值，则返回 NULL。

```
CRC32('xyz')                           → 3951999591
CRC32('0')                             → 4108050209
CRC32(0)                               → 4108050209
CRC32(NULL)                            → NULL
```

- ❑ DEGREES(*x*)

  返回值：弧度值 *x* 的角度值。

```
DEGREES(PI())           → 180
DEGREES(PI()*2)         → 360
DEGREES(PI()/2)         → 90
DEGREES(-PI())          → -180
```

- EXP(*x*)

  返回值：e^x（e 的 *x* 次方）。e 是自然对数的底数。

```
EXP(1)                  → 2.718281828459
EXP(2)                  → 7.3890560989307
EXP(-1)                 → 0.36787944117144
1/EXP(1)                → 0.36787944117144
```

- FLOOR(*x*)

  返回不大于 *x* 的最大整数。如果参数 *x* 是某种精确数值类型的，返回值也将有着同样的类型；否则，返回值将有着某种浮点（近似）数值类型，即使这个函数的返回值没有小数部分。

```
FLOOR(3.8)              → 3
FLOOR(-3.8)             → -4
```

- LN(*x*)

  本函数是 LOG(*x*) 函数的一个同义词。

- LOG(*x*)

  LOG(*b*, *x*)

  返回值：只带一个输入参数的 LOG(*x*) 函数将返回 *x* 以 e 为底的自然对数。

```
LOG(0)                  → NULL
LOG(1)                  → 0
LOG(2)                  → 0.69314718055995
LOG(EXP(1))             → 1
```

  带两个输入参数的 LOG(*b*, *x*) 函数将返回 *x* 以 *b* 为底的对数。

```
LOG(10,100)             → 2
LOG(2,256)              → 8
```

  你可以利用公式 LOG(*x*) / LOG(b) 来计算出 *x* 以 *b* 为底的对数。

```
LOG(100)/LOG(10)        → 2
LOG10(100)              → 2
```

- LOG10(*x*)

  返回值：*x* 以 10 为底的对数。

```
LOG10(0)                → NULL
LOG10(10)               → 1
LOG10(100)              → 2
```

- LOG2(*x*)

  返回值：*x* 以 2 为底的对数。

```
LOG2(0)                 → NULL
LOG2(255)               → 7.9943534368589
LOG2(32767)             → 14.99995597177
```

  LOG2() 能够让你了解以位计算的数据“宽度”。人们经常利用这个函数来估算某个值的存储空

间占用量。

- MOD(*m*, *n*)

  返回值：除法的余数。MOD(*m*, *n*)与 *m* % *n* 和 *m* MOD *n* 等价。另请参见 C.1.3 节。

- PI()

  返回值：圆周率 π。

```
PI()                                    → 3.141593
```

- POW(*x*, *y*)

  POWER(*x*, *y*)

  返回值：x^y，即 *x* 的 *y* 次方。

```
POW(2,3)                                → 8
POW(2,-3)                               → 0.125
POW(4,.5)                               → 2
POW(16,.25)                             → 2
```

- RADIANS(*x*)

  返回值：角度值 *x* 的弧度值。

```
RADIANS(0)                              → 0
RADIANS(360)                            → 6.2831853071796
RADIANS(-360)                           → -6.2831853071796
```

- RAND()

  RAND(*n*)

  RAND()函数将返回一个介于 0.0 和 1.0 之间的随机浮点值。如果有一个常整数参数 *n*，RAND(*n*)函数将以 *n* 作为随机数发生器的种子完成同样的事情。如果需要按照同样的顺序为结果集里的某个数据列生成一组重复的随机数，只要用同样的值作为种子就可以达到目的。（MySQL 5.0.13 及以后的版本禁止使用变量作为参数 *n*；在那之前，它们的影响未定义。）

```
RAND()                                  → 0.1036697114852
RAND()                                  → 0.5725383884949
RAND(10)                                → 0.65705152196535
RAND(10)                                → 0.65705152196535
```

  随机数发生器的种子值是针对具体客户（程序）的。某个客户调用 RAND(*n*)而提供给随机数发生器的种子值不会影响到其他客户调用 RAND()函数时得到的随机数。

  如果 RAND()出现在 WHERE 子句中，每次执行子句都会调用它。

- ROUND(*x*)

  ROUND(*x*, *d*)

  ROUND()函数将返回 *x* 的值，但只保留到小数点后面的 *d* 位数字。如果 *d* 等于零或没有给出，结果将没有小数点或小数部分。这个函数的返回值和它的第一个参数有着同样的数值类型，所以如果该参数是一个整数的话，返回结果将没有任何小数部分。以字符串形式给出的数值将先转换为双精度浮点数，再进行舍入处理。

```
ROUND(15.3)                             → 15
ROUND(15.5)                             → 16
ROUND(-33.27834,2)                      → -33.28
```

```
ROUND(1,4)                                  → 1
ROUND('1',4)                                → 1.0000
```

如果 *d* 是一个负数，ROUND(*x*, *d*)函数将去掉小数部分，并把从小数点开始往左算起的 ABS(d)位数字设置为零。

```
ROUND(123456,-2)                            → 123500
```

在 MySQL 5.0.3 之前的版本里，ROUND()函数的具体行为取决于其底层算术函数库所实现的舍入操作。这意味着 ROUND()函数的结果会随着系统平台的不同而变化。从 MySQL 5.0.3 版开始，ROUND()函数对 *x* 参数的舍入处理遵循以下规则。

- 对于近似值类型的数值，舍入行为仍取决于其底层的算术函数库。
- 对于精确值类型的数值，大于或等于 0.5 的小数部分将按远离零的方向舍入，小于 0.5 的小数部分将按接近零的方向舍入。比如说，1.5 和−1.5 将被分别舍入为 2 和−2，而 1.49 和−1.49 将被舍入为 1 和−1。

关于精确值类型和近似值类型的讨论，请参阅 3.1.1 节的第 1 小节。

- SIGN(*x*)

  返回值：根据 *x* 是负数、0、正数而分别返回−1、0、1。

```
SIGN(15.803)                                → 1
SIGN(0)                                     → 0
SIGN(-99)                                   → -1
```

- SIN(*x*)

  返回值：*x* 的正弦值。*x* 被视为一个弧度值。

```
SIN(0)                                      → 0
SIN(PI()/2)                                 → 1
```

- SQRT(*x*)

  返回值：*x* 的非负平方根。

```
SQRT(625)                                   → 25
SQRT(2.25)                                  → 1.5
SQRT(-1)                                    → NULL
```

- TAN(*x*)

  返回值：*x* 的正切值。*x* 被视为是一个弧度值。

```
TAN(0)                                      → 0
TAN(PI()/4)                                 → 1
```

- TRUNCATE(*x*, *d*)

  返回值：小数部分被截短为 *d* 位数字的 *x* 值。如果 *d* 等于 0，返回值里将不包含小数点和小数部分。如果 *d* 大于 *x* 的小数位数，*x* 的小数部分将用 0 来补足到指定的位数。

```
TRUNCATE(1.23,1)                            → 1.2
TRUNCATE(1.23,0)                            → 1
TRUNCATE(1.23,4)                            → 1.2300
```

如果 *d* 是负数，TRUNCATE()会去掉小数部分，把小数点左边的 ABS(*d*)个数变为 0。

```
TRUNCATE(123456.789,-3)                     → 123000
```

C.2.4 字符串函数

本节绝大多数字符串函数的返回值仍将是一个字符串。有些以字符串为输入参数的函数——比如 LENGTH()——却会返回一个数值。有一部分字符串类的函数是根据字符串位置处理的，我们将把字符串的头一个字符（字符串的最左端）称为第一个字符（而不是第 0 个字符）。

有些字符串函数能够支持多字节字符：CHAR_LENGTH()、INSERT()、INSTR()、LCASE()、LEFT()、LOCATE()、LOWER()、LTRIM()、MID()、POSITION()、REPLACE()、REVERSE()、RIGHT()、RPAD()、RTRIM()、SUBSTRING()、SUBSTRING_INDEX()、TRIM()、UCASE()和 UPPER()。

- ASCII(*str*)

  返回字符串 *str* 最左端的那个字符的 ASCII 编码，整数值，从 0 到 255。如果 *str* 是一个空字符串，则返回 0；如果 *str* 是 NULL，则返回 NULL。*str* 应该只包含 8 位的字符。

```
ASCII('abc')                                 → 97
ASCII('')                                    → 0
ASCII(NULL)                                  → NULL
```

- BIN(*n*)

  返回以字符串表示的数值 *n* 的二进制表示形式。下面两个表达式是等价的：

```
BIN(65)                                      → '1000001'
CONV(65,10,2)                                → '1000001'
```

  详细情况请参见对 CONV()函数的介绍。

- CHAR(*n1*, *n2*, ...[USING *charset*])

  在 MySQL 5.0.15 之前的版本里，CHAR()函数把参数 *n1*, *n2*,…解释为一组字符编码值，并根据它们用当前字符集里的对应字符构成一个字符串作为其返回值。字符编码值被解释为各参数值除以 256 的余数（只有最低字节的 8 个二进制位有效）。从 MySQL 5.0.15 版开始，大于 255 的参数值将被解释为多字节字符的字符编码，USING 选项也有了实际的用途。如果没有使用 USING 选项，CHAR()函数的返回值将是一个二进制字符串；如果使用了 USING 选项，返回值将由该选项所指定的字符集里的字符构成。如果某个参数值在指定字符集里没有与之对应的合法字符，CHAR()函数将返回一条警告消息(如果还启用了“严格”SQL 模式，结果将是 NULL)。NULL 参数将被忽略。

```
CHAR(65)                                     → 'A'
CHAR(97)                                     → 'a'
CHAR(89,105,107,101,115,33)                  → 'Yikes!'
```

- CHAR_LENGTH(*str*)

  CHARACTER_LENGTH(*str*)

  这两个函数与 LENGTH()差不多，但参数长度以字符计算，而非字节。(一个多字节字符的长度是 1。)

- CHARSET(*str*)

  返回给定字符串的字符集名字，如果参数非法，则返回 NULL。

```
CHARSET('abc')                               → 'latin1'
CHARSET(CONVERT('abc' USING utf8))           → 'utf8'
CHARSET(123)                                 → 'binary'
```

- COALESCE(*expr1*, *expr2*, ...)

  返回值：输入参数中的第一个非 NULL 元素。如果所有元素全都是 NULL，则返回 NULL。

```
COALESCE(NULL,1/0,2,'a',45+97)                  → '2'
COALESCE(NULL,1/0)                              → NULL
```

- COERCIBILITY (*str*)

  返回给定字符串 *str* 的可转换度（coercibility）如果该参数非法，返回 NULL。所谓“可转换度”是某给定字符串在涉及其他字符串的表达式里改变其排序方式的程度。表 C-3 按可转换度从低到高的顺序列出了这个函数可能的返回值。

表　C-3

可转换度	含　义
0	排序方式已明确设定，不能转换
1	未设定排序方式
2	排序方式是隐式设定的
3	排序方式由 USER()等系统值设定
4	排序方式是可转换的
5	排序方式是可忽略的（例如设置为 NULL）

```
COERCIBILITY(_utf8 'abc' COLLATE utf8_bin)      → 0
COERCIBILITY('abc')                             → 4
```

- COLLATION (*str*)

  返回给定字符串 str 的排序方式。如果该参数非法，返回 NULL。

```
COLLATION(_latin2 'abc')                        → 'latin2_general_ci'
COLLATION(CONVERT('abc' USING utf8) COLLATE utf8_bin)
                                                → 'utf8_bin'
```

- CONCAT (*str1*, *str2*, ...)

  返回一个由自身所有参数合并而成的字符串。只要有一个参数是 NULL，返回 NULL。只要有一个参数是二进制字符串，结果将是一个二进制字符串；如果所有参数都是非二进制字符串，结果将是一个非二进制字符串。每个数值类型的参数将被转换为一个二进制字符串，除非你明确地把它强制转换为一个非二进制字符串。CONCAT()函数允许只给出一个参数。

```
CONCAT('abc','def')                             → 'abcdef'
CONCAT('abc')                                   → 'abc'
CONCAT('abc',NULL)                              → NULL
CONCAT('Hello',', ','goodbye')                  → 'Hello, goodbye'
```

  合并字符串的另一个办法是相邻排列，即一个接一个地写出它们：

```
'three' 'blind' 'mice'                          → 'threeblindmice'
'abc' 'def' = 'abcdef'                          → 1
```

- CONCAT_WS(*delim*, *str1*, *str2*, ...)

  与 CONCAT()类似，返回值为由第 2 个及后续输入参数合并在一起而得到的一个字符串，各输入参数之间用字符串 delim 加以分隔。如果 delim 是 NULL，则返回 NULL；但参加合并的字符串里的 NULL 值和空字符串都将被忽略。

```
CONCAT_WS(',','a','b','','d')                    → 'a,b,,d'
CONCAT_WS('*-*','lemon','lime',NULL,'grape')     → 'lemon*-*lime*-*grape'
```

- CONV(*n*, *from_base*, *to_base*)

  返回值：把以 *from_base* 为底的数值 *n* 转换为以 *to_base* 为底，并把转换结果表示为一个字符串。只要参数中有 NULL 值，就将返回 NULL。参数 *from_base* 和 *to_base* 必须是一个 2 到 36 之间的整数。*n* 将被看做是一个 BIGINT（64 位）整数值，但不能被指定为一个字符串，因为底数大于 10 的数值可能会包含非十进制数字（这也是 CONV()函数返回一个字符串的原因。底数在 11 到 36 之间的数值转换而返回的字符可能是从 A 到 Z 的字母）。如果 *n* 不是一个合法的以 *from_base* 为底的数值，CONV()函数的返回值就将是 0。（比如说，如果 *from_base* 等于 16 而 *n* 是'abcdefg'，那么，因为 g 不是一个合法的十六进制数字，CONV()函数的返回值就将是 0。）

  数值 *n* 里的非十进制数字既可以是大写字母，也可以是小写字母；但返回值里的非十进制数字将全部写为大写字母。

  下面这个例子将把十六进制数 e（即十进制数 14）转换为二进制：

  ```
  CONV('e',16,2)                                   → '1110'
  ```

  下面这个例子将把二进制数 11111111（即十进制数 255）转换为八进制：

  ```
  CONV(11111111,2,8)                               → '377'
  CONV('11111111',2,8)                             → '377'
  ```

  在默认的情况下，数值 *n* 将被视为一个无符号数。但如果你给出的 *to_base* 是一个负值，函数 CONV()就将把 *n* 视为一个带符号的数值。如下所示：

  ```
  CONV(-10,10,16)                                  → 'FFFFFFFFFFFFFFF6'
  CONV(-10,10,-16)                                 → '-A'
  ```

- EXPORT_SET(*n*, *on*, *off*, [*delim*, [*bit_count*]])

  返回一个由子串 *on* 和 *off* 构成、以子串 *delim* 为分隔符的字符串。默认的分隔符是逗号。*on* 用来表示整数 *n* 中一个被置位（即等于 1）的位，*off* 用来表示整数 *n* 中没有被置位（即等于 0）的每位。结果中最左边的字符串对应于 *n* 中最下边的位。*bit_count* 是将对 *n* 值进行如此转换的最大比数。*bit_count* 的默认值是 64，这也是最大值。只要输入参数中有 NULL 值，这个函数的返回值就将是 NULL。

  ```
  EXPORT_SET(7,'+','-','',5)                       → '+++--'
  EXPORT_SET(0xa,'1','0','',6)                     → '010100'
  EXPORT_SET(97,'Y','N',',',8)                     → 'Y,N,N,N,N,Y,Y,N'
  ```

- FIND_IN_SET(*str*, *str_list*)

  返回值：*str_list* 是由一些以逗号分隔的子串（即类似于 MySQL 中的 SET 值）构成的一个字符串。FIND_IN_SET()将返回字符串 *str* 在 *str_list* 中的下标。如果 *str* 没有出现在 *str_list* 里，返回 0；只要输入参数中有 NULL 值，就返回 NULL。第一个子串的下标是 1。

  ```
  FIND_IN_SET('cow','moose,cow,pig')               → 2
  FIND_IN_SET('dog','moose,cow,pig')               → 0
  ```

- FORMAT(*x*, *d*)

  把数值 *x* 舍入到小数点后面第 *d* 位数字并写成“*nn,nnn.nnn*”的格式，返回值是一个字符串。

如果 *d* 等于 0，则返回值中将不包含小数点和小数部分。

```
FORMAT(1234.56789,3)                    → '1,234.568'
FORMAT(999999.99,2)                     → '999,999.99'
FORMAT(999999.99,0)                     → '1,000,000'
```

请注意最后一个例子里的数值舍入行为。

- HEX(*n*)

  HEX(*str*)

  如果输入参数是一个数值 *n*，HEX()函数将返回 *n* 的十六进制表示形式，返回值是一个字符串。下面两个表达式是等价的：

```
HEX(65)                                 → '41'
CONV(65,10,16)                          → '41'
```

  详细情况请参见对 CONV()函数的介绍。

  HEX()函数的输入参数可以是一个字符串，这时将把参数中的每一个字符转换为一个两位数的十六进制数，返回一个字符串：

```
HEX('255')                              → '323535'
HEX('abc')                              → '616263'
UNHEX(HEX('abc'))                       → 'abc'
```

- INSERT(*str*, *pos*, *len*, *ins_str*)

  把字符串 *str* 从第 *pos* 个位置开始的 *len* 个字符替换为 *ins_str* 后得到的一个字符串。如果 *pos* 超出字符串 *str* 的长度范围，则返回原来的字符串；只要输入参数中有 NULL 值，就返回 NULL。

```
INSERT('nighttime',6,4,'fall')          → 'nightfall'
INSERT('sunshine',1,3,'rain or')        → 'rain or shine'
INSERT('sunshine',0,3,'rain or')        → 'sunshine'
```

- INSTR(*str*, *substr*)

  INSTR()类似于带两个输入参数（但顺序却前后颠倒）的 LOCATE()函数。也就是说，下面两个表达式是等价的：

```
INSTR(str,substr)
LOCATE(substr,str)
```

- LCASE(*str*)

  本函数是 LOWER()函数的一个同义词。

- LEFT(*str*, *len*)

  返回字符串 *str* 最左面的 *len* 个字符，如果 *len* 大于 *str* 的长度，则返回整个字符串 *str*。如果 *str* 是 NULL，则返回 NULL；如果 *len* 是 NULL 或者小于 1，则返回一个空字符串。

```
LEFT('my left foot',2)                  → 'my'
LEFT(NULL,10)                           → NULL
LEFT('abc',NULL)                        → NULL
LEFT('abc',0)                           → ''
```

- LENGTH(*str*)

  返回字符串 *str* 的长度，以字节为单位。（多字节字符长度大于 1。）为确定字符长度，使用 CHAR_LENGTH()。

```
LENGTH('abc')                              → 3
LENGTH(CONVERT('abc' USING ucs2))          → 6
LENGTH('')                                 → 0
LENGTH(NULL)                               → NULL
```

- LOCATE(*substr*, *str*)

  LOCATE(*substr*, *str*, *pos*)

  只带两个输入参数的 LOCATE()函数将返回子串 *substr* 在字符串 *str* 里第一次出现位置的下标；如果子串 *substr* 没有出现在字符串 *str* 里，则返回 0。只要输入参数中有 NULL 值，就返回 NULL。如果还给出了一个位置参数 *pos*，LOCATE()函数将在字符串 *str* 里以 *pos* 为起点去寻找子串 *substr*。如果有一个操作数是二进制字符串，LOCATE()将按二进制字符串比较字符串；如果 *str* 和 *substr* 都不是二进制字符串，比较操作将按操作数排列方式进行。

```
LOCATE('b','abc')                          → 2
LOCATE('b','ABC')                          → 2
LOCATE(BINARY 'b','ABC')                   → 0
LOCATE('b' COLLATE latin1_general_ci,'ABC') → 2
LOCATE('b' COLLATE latin1_general_cs,'ABC') → 0
```

- LOWER(*str*)

  返回把字符串 *str* 里的字符全都转换为小写字母后得到的一个字符串。如果 *str* 是 NULL，则返回 NULL。

```
LOWER('New York, NY')                      → 'new york, ny'
LOWER(NULL)                                → NULL
```

  LOWER()函数是根据其参数字符集的排序方式来进行字母大小写转换的。如果它的参数是一个二进制字符串，LOWER()函数将原封不动地返回该参数本身，这是因为二进制字符串没有任何字符集和排序方式。

```
LOWER(BINARY 'New York, NY')               → 'New York, NY'
LOWER(0x414243)                            → 'ABC'
```

  如果这不是你想要的结果，可以把它的参数强制转换为一个有着适当排序方式的非二进制字符串：

```
LOWER(CONVERT(BINARY 'New York, NY' USING latin1))
                                           → 'new york, ny'
LOWER(_latin1 0x414243)                    → 'abc'
```

- LPAD(*str*, *len*, *pad_str*)

  返回在字符串 *str* 的左侧用子串 *pad_str* 补足到长度等于 *len* 个字符时得到的一个字符串。只要输入参数中有 NULL 值，就返回 NULL。

```
LPAD('abc',12,'def')                       → 'defdefdefabc'
LPAD('abc',10,'.')                         → '.......abc'
```

  如果字符串 *str* 的长度已经超过 *len* 个字符，LPAD()将把字符串 *str* 截短为 *len* 个字符：

```
LPAD('abc',2,'.')                          → 'ab'
```

- LTRIM(*str*)

  返回去掉字符串 *str* 最左端的一个字符后得到的那个字符串。如果 *str* 是 NULL，则返回 NULL。

```
LTRIM('abc')                                       → 'abc'
```

- MAKE_SET(*n*, *bit0_str*, *bit1_str*, ...)

根据整数值 *n* 和子串 *bit0_str*、*bit1_str*...而构造出来的一个 *SET* 值（即一个以逗号来分隔各个子串的字符串）。*n* 值中每一个被置位（即等于 1）的位所对应的子串都将被包括在返回值里。（比如说，如果 *n* 值的位被置位，*bit0_str* 就会被包括在结果里。）如果 *n* 等于 0，则返回一个空字符串；如果 *n* 是 NULL，则返回 NULL。如果有任何字符串为 NULL，在构造结果字符串时它将被忽略。

```
MAKE_SET(8,'a','b','c','d','e')                    → 'd'
MAKE_SET(1|2|4,'a','b','c','d','e')                → 'a,b,c'
MAKE_SET(2+16,'a','b','c','d','e')                 → 'b,e'
MAKE_SET(-1,'a','b','c','d','e')                   → 'a,b,c,d,e'
```

在最后一个例子里，因为 *n* 等于−1，所以返回值将包含每一个子串。

- MATCH(*col_list*) AGAINST(*str* [*search_mode*])

MATCH(*col_list*) AGAINST(*str* IN BOOLEAN MODE)

MATCH(*col_list*) AGAINST(*str* WITH QUERY EXPANSION)

MATCH()将使用一个 FULLTEXT 索引来进行一次搜索操作。*column_list* 是由一个或者多个数据列的名字构成的列表，各数据列的名字以逗号分隔；而将被搜索的数据表里也必须有一个由这些数据列构成的 FULLTEXT 索引。AGAINST(*str*)里的 *str* 是你想在这些数据列里查找的一个或者多个单词，单词是由字母、数字、单引号或者下划线字符构成的字符序列。MATCH 中允许出现括号，但 AGAINST 中不允许出现括号。

默认情况下，以自然语言模式执行搜索。显式的 *search_mode* 参数可以有以下其中一个值。

- IN NATURAL LANGUAGE MODE
- IN BOOLEAN MODE
- WITH QUERY EXPANSION
- IN NATURAL LANGUAGE MODE WITH QUERY EXPANSION

包含 IN NATURAL LANGUAGE MODE 的模式在 MySQL 5.1.7 中引入。

对于自然语言搜索，MATCH()给出的是被搜索单词在每个数据行里的相关度。这些值都是非负的浮点数：零分布统计值表示在数据表里没有找到被搜索单词，正分布统计值则表明至少找到一个被搜索单词。如果在数据表一半以上的数据行里都找到过被搜索单词，它们的相关度将被认为是零，因为它们的出现次数太多了。此外，MySQL 内部还有一个停止单词（比如“the”和“but”）清单，其中单词的相关度为 0。

如果搜索模式是 IN BOOLEAN MODE，那么搜索结果将以被搜索单词是否出现过为依据，而不是以它们的出现频率为依据。在布尔搜索方式中，你还可以通过给被搜索单词加上以下几种修饰符来影响搜索操作的具体行为。

- 出现在被搜索单词前面的加号（+）或减号（-）表示该单词必须出现或必须不出现。
- 出现在被搜索单词前面的小于号（<）或大于号（>）将削弱或增强该单词对相关度值计算结果的贡献。
- 一个前导的“~”字符将使给定单词在相关度计算公式里的贡献值变换正负号，但不会像“-”前导字符那样把包含该单词的数据行完全排除在外。

- 出现在被搜索单词尾部的星号字符（*）被看做是一个通配符。比如说，“act*”将匹配“act”、“acts”、“action”等。
- 短语必须用双引号括起来。"phase"形式的短语搜索必须在有关单词的排列顺序也完全一致时才算匹配成功。
- 多个被搜索单词可以用括号归组为一个表达式。括号表达式可以嵌套。

在布尔搜索方式中，那些不带修饰符的被搜索单词都是可选的，与普通搜索方式中的含义相同。布尔搜索方式在数据表没有相应的 FULLTEXT 索引时也能进行，但速度往往会非常的慢。

如果搜索模式是 WITH QUERY EXPANSION，只要使用搜索字符串进行搜索，然后使用搜索字符串和与初次搜索最为匹配的少量信息再次搜索，自然语言搜索就完成了。这样能找出内容与初次搜索字符串相关的数据行。

有关 FULLTEXT 搜索机制的信息参见 2.15 节。

- MID(*str*, *pos*, *len*)

  MID(*str*, *pos*)

  MID(*str*, *pos*, *len*)将返回字符串 *str* 从位置 *pos* 开始且长度为 *len* 个字符的那一个子串。MID(*str*, *pos*)将返回字符串 *str* 从位置 *pos* 到最后一个字符的那一个子串。只要输入参数中有 NULL 值，就返回 NULL。

```
MID('what a dull example',8,4)                  → 'dull'
MID('what a dull example',8)                    → 'dull example'
```

  事实上，MID()是 SUBSTRING()函数的一个同义词。SUBSTRING()函数的各种语法形式都能用在 MID()函数里。

- OCT(*n*)

  返回包含数值 *n* 的八进制表示形式的一个字符串。下面两个表达式是等价的：

```
OCT(65)                                         → '101'
CONV(65,10,8)                                   → '101'
```

  详细情况请参见对 CONV()函数的介绍。

- OCTET_LENGTH(*str*)

  本函数是 LENGTH()函数的一个同义词。

- ORD(*str*)

  返回字符串 *str* 第一个字符的排位序号，如果 *str* 是 NULL，则返回 NULL。如果第一个字符是一个单字节字符，ORD()函数将等价于 ASCII()函数。

```
ORD('abc')                                      → 97
ASCII('abc')                                    → 97
```

  如果 *str* 参数是一个多字节字符串，ORD()函数将按以下公式计算其第一个字符的序号值（假设该字符的各个字节按从右至左的顺序依次是 *b1*, *b2*, ... *bn*）：

```
b1 + (b2 × 256) + (b3 × 256 × 256) + ...
```

- POSITION(*substr* IN *str*)

  本函数相当于只带两个输入参数的 LOCATE()函数。下面两个表达式是等价的：

```
POSITION(substr IN str)
LOCATE(substr, str)
```

- QUOTE(*str*)

按 SQL 语句的使用要求在输入参数 *str* 里正确地添加各种引号后得到的一个字符串。这个函数在编写能生成其他查询语句的查询语句时非常有用。对于非 NULL 值，返回值有一个单引号、反斜线、Ctrl-Z 字符和一个以反斜线（\）字符转义，结果用单引号的 NULL（0 值字节）括起来。对于 NULL 值输入参数 *str*，返回值是不带单引号的单词 NULL。如下所示：

```
QUOTE("Let's go!")                              → 'Let\'s go!'
QUOTE(NULL)                                     → 'NULL'
```

- REPEAT(*str*, *n*)

  返回值是把字符串 *str* 重复 *n* 次后得到的一个字符串。如果 *n* 是负数或零，则返回一个空字符串。只要输入参数中有 NULL 值，就返回 NULL。

```
REPEAT('x',10)                                  → 'xxxxxxxxxx'
REPEAT('abc',3)                                 → 'abcabcabc'
```

- REPLACE(*str*, *from_str*, *to_str*)

  返回值是把字符串 *str* 中的子串 *from_str* 全部替换为 *to_str* 后得到的一个字符串。如果 *to_str* 是空字符串，则效果相当于把 *from_str* 全都去掉。如果 *from_str* 是空字符串，REPLACE()将不对字符串 *str* 做任何改变。只要输入参数中有 NULL 值，就返回 NULL。

```
REPLACE('abracadabra','a','oh')                 → 'ohbrohcohdohbroh'
REPLACE('abracadabra','a','')                   → 'brcdbr'
REPLACE('abracadabra','','x')                   → 'abracadabra'
```

- REVERSE(*str*)

  返回值是前后颠倒字符串 *str* 里的所有字符后得到的一个字符串。若 *str* 是 NULL，则返回 NULL。

```
REVERSE('abracadabra')                          → 'arbadacarba'
REVERSE('tararA ta tar a raT')                  → 'Tar a rat at Ararat'
```

- RIGHT(*str*, *len*)

  返回值：字符串 *str* 最右面的 *len* 个字符，如果 *len* 大于 *str* 的长度，则返回整个字符串 str。如果 *str* 是 NULL，则返回 NULL；如果 *len* 是 NULL 或者小于 1，则返回一个空字符串。

```
RIGHT('rightmost',4)                            → 'most'
```

- RPAD(*str*, *len*, *pad_str*)

  这个函数的返回值是通过尾缀 *pad_str* 把字符串 *str* 的长度补足到 *len* 个字符后得到的一个字符串。只要它的参数里有一个是 NULL，RPAD()函数将返回 NULL。

```
RPAD('abc',12,'def')                            → 'abcdefdefdef'
RPAD('abc',10,'.')                              → 'abc.......'
```

  如果字符串 *str* 的长度已经超过 len 个字符，RPAD()将把字符串 *str* 截短为 *len* 个字符：

```
RPAD('abc',2,'.')                               → 'ab'
```

- RTRIM(*str*)

  返回值：去掉字符串 *str* 最右端的空格后得到的那个字符串。如果 str 是 NULL，则返回 NULL。

```
RTRIM('abc')                                    → 'abc'
```

- SOUNDEX(*str*)

  expr1 SOUNDS LIKE *expr2*

返回值：根据字符串 *str* 计算出来的一个 soundex 字符串，如果 *str* 是 NULL，则返回 NULL。字符串 *str* 中不是字母或数字的字符都将被忽略。不在从 A 到 Z 范围以内的非字母国际字符都将被视为元音。对于有多字节字符的字符串或英语之外的其他语言，SOUNDEX()结果可能是无意义的。如下所示：

```
SOUNDEX('Cow')                              → 'C000'
SOUNDEX('Cowl')                             → 'C400'
SOUNDEX('Howl')                             → 'H400'
SOUNDEX('Hello')                            → 'H400'
```

SOUNDS LIKE 操作符等价于 SOUNDEX()函数。

- SPACE(*n*)

  返回值：一个由 *n* 个空格构成的字符串。如果 *n* 不是一个正数，则返回一个空字符串；如果 *n* 是 NULL，则返回 NULL。

```
SPACE(6)                                    → '      '
SPACE(0)                                    → ''
SPACE(NULL)                                 → NULL
```

- SUBSTR(*arguments*)

  SUBSTR()是 SUBSTRING()的同义词，参数格式可以一样。

- SUBSTRING(*str*,*pos*)

  SUBSTRING(*str*,*pos*,*len*)

  SUBSTRING(*str* FROM *pos*)

  SUBSTRING(*str* FROM *pos* FOR *len*)

  返回值：字符串 *str* 从位置 *pos* 开始的一个子串。只要输入参数中有 NULL 值，就返回 NULL。如果给出了 *len* 参数，则作为返回值子串的长度将是 *len* 个字符；否则，将返回字符串 *str* 从位置 *pos* 开始直到最后一个字符的子串。

```
SUBSTRING('abcdef',3)                       → 'cdef'
SUBSTRING('abcdef',3,2)                     → 'cd'
```

下面这几个表达式都是等价的：

```
SUBSTRING(str,pos,len)
SUBSTRING(str FROM pos FOR len)
MID(str,pos,len)
```

- SUBSTRING_INDEX(*str*, *delim*, *n*)

  返回按以下规则得到的字符串 *str* 的一个子串：（1）如果 *n* 是正值，SUBSTRING_INDEX()函数将按从左向右的顺序找到子串 *delim* 的第 *n* 次出现并返回该位置左侧的全部内容；（2）如果 *n* 是负值，SUBSTRING_INDEX()函数将按从右向左的顺序找到子串 *delim* 的第 *n* 次出现并返回该位置右侧的全部内容；（3）如果没有在字符串 *str* 里找到子串 *delim*，则原样返回字符串 *str*；（4）只要输入参数中有 NULL 值，就返回 NULL。

```
SUBSTRING_INDEX('jar-jar','j',-2)           → 'ar-jar'
SUBSTRING_INDEX('sampadm@localhost','@',1)  → 'sampadm'
SUBSTRING_INDEX('sampadm@localhost','@',-1) → 'localhost'
```

- TRIM([*trim_str* FROM] *str*)

  TRIM([{LEADING | TRAILING | BOTH} [*trim_str*] FROM] *str*)

返回按以下规则在字符串 *str* 的首/尾去掉子串 *trim_str* 后得到的一个字符串：(1) 如果给出了 LEADING，TRIM()函数将去掉字符串 *str* 最前（左）端的子串 *trim_str*；(2) 如果给出了 TRAILING，TRIM()函数将去掉字符串 *str* 最尾（右）端的子串 *trim_str*；(3) 如果给出了 BOTH，TRIM()函数将去掉字符串 *str* 前后（左、右）两端的子串 *trim_str*；(4) 如果 LEADING、TRAILING 或 BOTH 都没有给出，则按 BOTH 情况做默认处理；(5) 如果没有给定子串 *trim_str*，TRIM()函数将去掉空格。如下所示：

```
TRIM('^' FROM '^^^xyz^^')                          → 'xyz'
TRIM(LEADING '^' FROM '^^^xyz^^')                  → 'xyz^^'
TRIM(TRAILING '^' FROM '^^^xyz^^')                 → '^^^xyz'
TRIM(BOTH '^' FROM '^^^xyz^^')                     → 'xyz'
TRIM(BOTH FROM 'abc')                              → 'abc'
TRIM('abc')                                        → 'abc'
```

- UCASE(*str*)

  本函数是 UPPER()函数的一个同义词。

- UNHEX(*expr*)

  参数被解释为包含几对十六进制数字的字符串。每对数字被转换为一个字符，返回值是一个由这些字符组成的二进制字符串。UNHEX()是 HEX()的反转结果。

```
UNHEX('414243')                                    → 'ABC'
HEX(UNHEX('414243'))                               → '414243'
UNHEX(HEX('ABC'))                                  → 'ABC'
UNHEX(414243)                                      → 'ABC'
CHARSET(UNHEX('414243'))                           → 'binary'
```

- UPPER(*str*)

  返回值：把字符串 *str* 里的字符全都转换为大写字母后得到的一个字符串。如果 *str* 是 NULL，则返回 NULL。

```
UPPER('New York, NY')                              → 'NEW YORK, NY'
UPPER(NULL)                                        → NULL
```

  关于二进制字符串的字母大小写转换问题，请参阅前面对 LOWER()函数的描述。

- WEIGHT_STRING (*str* [AS *type*(*n*)] [LEVEL *levels*] [*flags*])

  以二进制字符串的形式返回 *str* 参数的权重字符串。权重字符串是 MySQL 在进行字符串比较和排序操作的时候使用的一种内部表示形式。两个权重字符串相同的字符串在比较操作中将被认为是相等的，或者更准确地说，它们有着与它们的权重字符串相同的相对顺序。这个函数的 AS 选项用来把 *str* 参数转换为某种给定的类型和长度，LEVEL 选项用来指定返回哪一级排序级别的权重字符串。*flags* 值在现阶段还没有任何具体的实现。

  如果 *str* 参数是一个二进制字符串，其权重字符串将与 *str* 相同。如果 *str* 参数是一个非二进制字符串，它就会有一种排序方式，而它的权重字符串也将包含着该种排序方式的权重。如果 *str* 参数是 NULL，这个函数的返回结果将是 NULL。下面几个例子为了把权重字符串显示为可打印格式而使用了 HEX()函数：

```
HEX(WEIGHT_STRING(BINARY 'Hello'))                 → '48656C6C6F'
HEX(WEIGHT_STRING('Hello'))                        → '48454C4C4F'
HEX(WEIGHT_STRING(_utf8'Hello'))                   → '00480045004C004C004F'
```

str 参数可以用 AS 子句强制转换为某种给定的类型和长度。比如说，AS CHAR(*n*)将把 *str* 强制转换为一个长度是 *n* 个字符的 CHAR 字符串。如有必要，在该字符串的尾部用空格补齐。AS BINARY(*n*)将把 *str* 强制转换为一个长度是 *n* 个字节的二进制字符串，必要时用零值字节（0x00）补齐。*n* 值必须大于或等于 1。如果它的长度大于 *n* 个字符或字节，*str* 将被截短而不是被补齐。

一种给定的排序方式可能会有多个级别。在默认的情况下，这个函数的返回结果将包含所有级别的权重。如果只想返回某几个特定级别的权重，就需要用到 LEVEL 选项。levels 参数值既可以是一列由几个以逗号分隔的整数所构成的值，也可以是一个由两个以短划线字符分隔的整数所设定的区间。各个级别必须按递增的顺序给出。区间里的第二个级别如果小于第一个级别将被视为等于第一个级别。排序方式的级别从 1 开始，并有一个最大值，超出这个范围的 *levels* 参数值将按关规则被换算成这个范围内的某个值。

每个级别值的后面还可以给出一个限定符：ASC，返回未做任何修改的权重（默认情况）；DESC，返回逐位反转的权重；REVERSE，返回把 str 前后颠倒而得到的那个字符串值的权重。

```
HEX(WEIGHT_STRING('abc' LEVEL 1 ASC))          → '414243'
HEX(WEIGHT_STRING('abc' LEVEL 1 DESC))         → 'BEBDBC'
HEX(WEIGHT_STRING('abc' LEVEL 1 REVERSE))      → '434241'
```

WEIGHT_STRING()函数是从 MySQL 5.2.4 版开始引入的。

C.2.5 日期/时间类函数

日期/时间类函数往往允许输入参数是多种类型。一般说来，接受 DATE 值作为输入参数的函数通常也接受 DATETIME 或 TIMASTAMP 值作为其参数并忽略其中的时间部分；而接受 TIME 值作为输入参数的函数通常也接受 DATETIME 或 TIMASTAMP 值作为输入参数并忽略其中的日期部分。

本节中的大部分函数都能把数值形式的输入参数解释为日期/时间值，如下所示：

```
MONTH('2008-07-25')                              → 7
MONTH(20080725)                                  → 7
```

类似地，那些返回值原本是一个日期/时间值的函数也大都能根据上下文将返回值转换为一个字符串或一个数值，如下所示：

```
CURDATE()                                        → '2008-05-01'
CONCAT('Today is ', CURDATE())                   → 'Today is 2008-05-01'
CURDATE() + 0                                    → 20080501
```

在需要把时间值或日期/时间值转换为数值的时候，转换结果将有一个“.000000”形式的微秒部分。如果想去掉这个部分，把转换结果强制转换为一个整数即可：

```
NOW()+0                                          → 20080501183210.000000
CURTIME()+0                                      → 183210.000000
CAST(NOW() AS UNSIGNED)                          → 20080501183210
CAST(CURTIME() AS UNSIGNED)                      → 183210
```

有几个用来提取日期部分的函数，在遇到“不完整的”日期时会返回 0。比如说，MONTH()和 DAYOFMONTH()函数在遇到参数'2013-00-00'的时候都将返回 0。用在 DATE_FORMAT()函数里的日期部分的格式限定符也是如此。

如果提供给日期/时间函数的日期值或时间值不合法，就不能指望能够得到合理的结果。一定要提

前检查参数。

- ADDDATE (*date*, *INTREVAL expr interval*)
  ADDDATE (*date*, *expr*)
  对于第一种语法，ADDDATE()函数通过 *date* 参数读入一个日期值或一个日期/时间值，给它加上一段时间间隔，最后返回计算的结果。它是 ADD_DATE()函数的一个同义词：

```
ADDDATE('2004-12-01',INTERVAL 1 YEAR)          → '2005-12-01'
```

  对于第二种语法，ADDDATE()函数通过 date 参数读入一个日期值或一个日期/时间值，给它加上一个天数，最后返回计算的结果。

```
ADDDATE('2004-12-01',365)                      → '2005-12-01'
```

  第二种语法可以改写为第一种语法，如下所示：

```
ADDDATE(date,expr) = ADDDATE(date,INTERVAL expr DAY)
```

- ADDTIME(*expr1*,*expr2*)
  把两个表达式相加并返回其结果。*expr1* 应该是一个时间值或一个日期/时间值，*expr2* 应该是一个时间值。两个值都可以包含微秒部分。

```
ADDTIME('06:30:00.5','12:30:00.5')              → '19:00:01.000000'
ADDTIME('2004-01-01 00:00:00','12:30:00')       → '2004-01-01 12:30:00'
```

- CONVERT_TZ(*date*, *from_zone*, *to_zone*)
  给定日期值或日期/时间值 *date*，CONVERT_TZ ()函数把它视为 *from_zone* 时区里的一个值，把它转化为 *to_zone* 时区里的一个值，最后返回转换结果。只要这些参数里有一个是非法的，将返回 NULL。时区可以按 12.9.1 节里介绍的办法给出。为了让这个函数正确工作，转换结果必须落在 TIMESTAMP 数据类型的表示范围内。

```
CONVERT_TZ('2009-02-11 00:00:00','US/Central','US/Eastern')
                                                → '2009-02-11 01:00:00'
CONVERT_TZ('2009-02-11','+00:00','-03:00')      → '2009-02-10 21:00:00'
```

- CURDATE()
  返回值：当前日期，是一个'*CCYY-MM-DD*'格式的日期值。

```
CURDATE()                                       → '2008-05-01'
```

- CURRENT_DATE()
  本函数是 CURDATE()函数的一个同义词，括号可选。
- CURRENT_TIME()
  本函数是 CURTIME()函数的一个同义词，括号可选。
- CURRENT_TIMESTAMP()
  本函数是 NOW()函数的一个同义词，括号可选。
- CURTIME()
  返回值：当前时间，是一个'hh:*mm*:*ss*'格式的时间值。

```
CURTIME()                                      → '18:32:58'
```

- DATE(*expr*)
  返回 *expr* 的日期部分，应该是一个日期表达式或日期加时间表达式。

```
DATE('2008-03-12')                                → '2008-03-12'
DATE('2008-03-12 16:15:00')                       → '2008-03-12'
```

- DATE_ADD(date, *INTERVAL expr interval*)

返回值：日期值或日期/时间值 *date* 加上一段时间间隔后得到的一个日期/时间值。其中，表达式 *expr* 给出了将与 *date* 值进行加减（若表达式 *expr* 以负号“-”开头，则减去）的数值，时间类型标识符 interval 则表明了这段时间间隔的解释办法。如果 *date* 是一个 DATE 值，则返回值也将是一个 DATE 值，有关参数中的时间部分不参加计算；否则，返回值将是一个 DATETIME 值。如果 *date* 不是一个合法的日期/时间值，DATE_ADD()函数将返回 NULL。

```
DATE_ADD('2009-12-01',INTERVAL 1 YEAR)            → '2010-12-01'
DATE_ADD('2009-12-01',INTERVAL 60 DAY)            → '2010-01-30'
DATE_ADD('2009-12-01',INTERVAL -3 MONTH)          → '2009-09-01'
DATE_ADD('2009-12-01 08:30:00',INTERVAL 12 HOUR)  → '2009-12-01 20:30:00'
```

表 C-4 列出了 *interval* 的可取值、含义以及使用格式。关键字 INTERVAL 和 interval 不区分大小写。

表 C-4

Interval类型	含 义	值的格式
MICROSECOND	微秒	*uuuuuu*
SECOND	秒	*ss*
SECOND_MICROSECOND	秒和微秒	*'ss.uuuuuu'*
MINUTE	分	*mm*
MINUTE_SECOND	分和秒	*'mm:ss'*
MINUTE_MICROSECOND	分和微秒	*'mm.uuuuuu'*
HOUR	小时	*hh*
HOUR_MINUTE	小时和分	*'hh:mm'*
HOUR_SECOND	小时、分和秒	*'hh:mm:ss'*
HOUR_MICROSECOND	小时和微秒	*'hh.uuuuuu'*
DAY	天数	*DD*
DAY_HOUR	天数和小时	*'DD hh'*
DAY_MINUTE	天数、小时和分	*'DD hh:mm'*
DAY_SECOND	天数、小时、分和秒	*'DD hh:mm:ss'*
DAY_MICROSECOND	天和微秒	*'DD.uuuuuu'*
WEEK	周	*WW*
MONTH	月	*MM*
QUARTER	季度	*QQ*
YEAR	年	*YY*
YEAR_MONTH	年和月	*'YY-MM'*

添加到日期的表达式 *expr* 既可以指定为一个数值，也可以指定为一个字符串；但如果它里面包含有非数字的字符，那就必须指定为一个字符串。表达式 *expr* 中的间隔符允许是任何一种标点符号。

```
DATE_ADD('2005-12-01',INTERVAL '2:3' YEAR_MONTH)  → '2008-03-01'
DATE_ADD('2005-12-01',INTERVAL '2-3' YEAR_MONTH)  → '2008-03-01'
```

表达式 *expr* 的各组成部分将按 *interval* 限定的输入格式从右向左进行匹配和解释。比如说，HOUR_SECOND 限定的输入格式是'*hh*:*mm*:*ss*'，如果表达式 *expr* 的值是'15:21'，那它将被解释为'00:15:21'而不是'15:21:00'。

```
DATE_ADD('2003-12-01 12:00:00',INTERVAL '15:21' HOUR_SECOND)
                                                  → '2003-12-01 12:15:21'
```

如果 *interval* 是 YEAR、MONTH 或 YEAR_MONTH，而计算结果中的天数部分却大于当月的天数，DATE_ADD()函数将把天数自动调整为当月的最大日期值，如下所示：

```
DATE_ADD('2003-12-31',INTERVAL 2 MONTH)           → '2004-02-29'
```

日期加法还支持使用下面这种语法格式：

```
'2003-12-31' + INTERVAL 2 MONTH                   → '2004-02-29'
INTERVAL 2 MONTH + '2003-12-31'                   → '2004-02-29'
```

- DATE_FORMAT(*date*, *format*)

  返回值：按格式字符串 *format* 对日期或日期/时间值 *date* 进行格式化后得到一个字符串。这个函数能把 MySQL 所支持的各种 DATE 和 DATETIME 格式重新编排为给定的格式。

```
DATE_FORMAT('2004-12-01','%M %e, %Y')             → 'December 1, 2004'
DATE_FORMAT('2004-12-01','The %D of %M')          → 'The 1st of December'
```

  下面的表格列出了允许用在格式字符串里的限定符。月、日限定符的数值表示范围从零开始，因为不完整的日期需要用零来表示其月份或日子，如'2004-00-13'或'1998-12-00'。

  每个格式限定符前面的%字符是必不可少的。格式字符串里包含的没在表 C-5 里列出的字符将原封不动地显示在输出结果里。

表 C-5

格式说明符	含 义
%f	以六位数字表示的微秒值（000000, 000001，…）
%S 或 %s	以两位数字表示的秒值（00, 01, …, 59）
%i	以两位数字表示的分钟值（00, 01, …, 59）
%H	以两位数字表示的小时值，24小时制（00, 01, …, 23）
%h 或 %I	以两位数字表示的小时值，12小时制（00, 01, …, 12）
%k	以数值表示的小时值，24小时制（0, 1, …, 23）
%l	以数值表示的小时值，12小时制（0, 1, …, 12）
%T	24小时制的时间值（*hh*:*mm*:*ss*）
%r	12小时制的时间值（*hh*:*mm*:*ss* AM或者*hh*:*mm*:*ss* PM）
%p	AM或者PM
%W	星期几（Sunday、Monday、…, Saturday）
%a	星期几的简写形式（Sun、Mon、… Sat）
%d	以两位数字表示的日期（00, 01, …, 31）
%e	以数值表示的日期（0, 1, 2, … , 31）
%D	英语字做后缀的日期（oth, 1st, 2nd, 3rd, …）

（续）

格式说明符	含 义
%w	以数值表示的星期几（0=Sunday, 1=Monday, ⋯ , 6=Saturday）
%j	以三位数字表示的一年中的天数（001, 002, ⋯, 366）
%U	一年中的第几个星期（00, ⋯, 53），以Sunday（星期日）作为每星期的第一天
%u	一年中的第几个星期（00, ⋯, 53），以Monday（星期一）作为每星期的第一天
%V	一年中的第几个星期（01, ⋯ , 53），以Sunday（星期日）作为每星期的第一天
%v	一年中的第几个星期（01, ⋯ , 53），以Monday（星期一）作为每星期的第一天
%M	月份值（January、February, ⋯ , December）
%b	月份值的简写形式（Jan, Feb, ⋯, Dec）
%m	以两位数字表示的月份值（00, 01, 02, ⋯, 12）
%c	以数值表示的月份值（0, 1, 2, ⋯ , 12）
%Y	以四位数字表示的年份值
%y	以两位数字表示的年份值
%X	以Sunday（星期日）作为每星期第一天的年份值，以四位数字表示
%x	以Monday（星期一）作为每星期第一天的年份值，以四位数字表示
%%	%字符本身

如果对 DATE 值使用时间说明符，值的时间部分就被当做'00:00:00'。

```
DATE_FORMAT('2004-12-01','%i')                          → '00'
```

- DATE_SUB(*date*, *INTERVAL expr interval*)

  返回值：按与 DATE_ADD()函数同样的规则计算出来日期/时间值。只不过 DATE_SUB()函数是对表达式 *expr* 和日期或日期/时间值 *date* 做减法。详细情况请参见 DATE_ADD()条目。

```
DATE_SUB('2009-12-01',INTERVAL 1 MONTH)             → '2009-11-01'
DATE_SUB('2009-12-01',INTERVAL '13-2' YEAR_MONTH)   → '1996-10-01'
DATE_SUB('2009-12-01 04:53:12',INTERVAL '13-2' MINUTE_SECOND)
                                                    → '2009-12-01 04:40:10'
DATE_SUB('2009-12-01 04:53:12',INTERVAL '13-2' HOUR_MINUTE)
                                                    → '2009-11-30 15:51:12'
```

  日期减法还支持使用下面这种语法格式：

```
'2009-12-01' - INTERVAL 1 MONTH                     → '2009-11-01'
```

  使用这种语法时，必须把 INTERVAL 子句放在减法操作符的右侧，因为用一段“时间间隔”去减一个日期没有道理。

- DATEDIFF (*expr1*, *expr2*)

  计算并返回两个表达式相距的天数，两个表达式都应该是日期值或日期/时间值。如果第一个参数给出的时间晚于第二个，这个函数的返回值将是一个正数。两个参数里的时间部分都将被忽略。

```
DATEDIFF('1987-01-01','1987-01-08')                 → -7
DATEDIFF('1987-01-08','1987-01-01')                 → 7
DATEDIFF('1987-01-01 12:00:00','1987-01-08')        → -7
DATEDIFF('1987-01-08','1987-01-01 12:00:00')        → 7
```

- DAY (*date*)

  这个函数是 DAYMONTH()函数的同义词。

- DAYNAME(*date*)

  返回值：日期/时间值 *date* 是星期几，是字符串形式。如果名字不确定，则为 NULL。

```
DAYNAME('2004-12-01')                      → 'Wednesday'
DAYNAME('1900-12-01')                      → 'Saturday'
DAYNAME('1900-12-00')                      → NULL
```

- DAYOFMONTH(*date*)

  返回值：日期/时间值 *date* 是几号，为数值形式，从 0 到 31（为 0 时，表示没有“日”部分）。

```
DAYOFMONTH('2002-12-01')                   → 1
DAYOFMONTH('2002-12-25')                   → 25
DAYOFMONTH('2002-12-00')                   → 0
```

- DAYOFWEEK(*date*)

  返回值：日期值 *date* 是星期几，为数值形式。按照 ODBC 标准的规定，星期几的范围将是 1 到 7，其中 1 代表 Sunday（星期日）、7 代表 Saturday（星期六）。另请参见 WEEKDAY()条目。

```
DAYOFWEEK('2004-12-05')                    → 1
DAYNAME('2004-12-05')                      → 'Sunday'
DAYOFWEEK('2004-12-18')                    → 7
DAYNAME('2004-12-18')                      → 'Saturday'
```

- DAYOFYEAR(*date*)

  返回值：日期值 *date* 是一年中的第几天，取值范围是 1 到 366。

```
DAYOFYEAR('2002-12-01')                    → 335
DAYOFYEAR('2004-12-31')                    → 366
```

- EXTRACT(*interval* FROM *datetime*)

  返回值：从日期/时间值 *datetime* 里根据 *interval*（可以是 DATE_ADD()允许的任何一个）而截取出来的那个子值。

```
EXTRACT(YEAR FROM '2002-12-01 13:42:19')           → 2002
EXTRACT(MONTH FROM '2002-12-01 13:42:19')          → 12
EXTRACT(DAY FROM '2002-12-01 13:42:19')            → 1
EXTRACT(HOUR_MINUTE FROM '2002-12-01 13:42:19')    → 1342
EXTRACT(SECOND FROM '2002-12-01 13:42:19')         → 19
```

  EXTRACT()函数还能对有“残缺”的日期值进行截取：

```
EXTRACT(YEAR FROM '2004-00-12')            → 2004
EXTRACT(MONTH FROM '2004-00-12')           → 0
EXTRACT(DAY FROM '2004-00-12')             → 12
```

- FROM_DAYS(*n*)

  把从公元 0 年 1 月 1 日开始计算的天数 *n*（通常是调用 TO_DAYS()函数后的返回值）转换为相应的日期。

```
TO_DAYS('2009-12-01')                      → 734107
FROM_DAYS(734107 + 3)                      → '2009-12-04'
```

  FROM_DAYS()函数只能转换出格雷果里历法日期（自公元 1582 年以来西方国家使用的历法）。

- FROM_UNIXTIME(*unix_timestamp*)

  FROM_UNIXTIME(*unix_timestamp*,*format*)

  把给定的 Unix 时间戳值 *unix_timestamp*（如 UNIX_TIMESTAMP()函数的返回值）转换为当前地理时区中的一个'*CCYY-MM-DD hh*:*mm*:*ss*'格式的 DATETIME 值。如果还给出了 *format* 参数，返回值将按 *format* 参数在 DATE_FORMAT()函数里的释义被排版为一个字符串。

```
UNIX_TIMESTAMP()                                  → 1209684883
FROM_UNIXTIME(1209684883)                         → '2008-05-01 18:34:43'
FROM_UNIXTIME(1209684883,'%Y')                    → '2008'
```

- GET_FORMAT (*val_type*, *format_type*)

  根据 *format_type* 参数的要求把 *val_type* 参数转换为一个可以用在 DATE_FORMAT()、TIME_FORMAT()和 STR_TO_DATE()函数里的格式字符串。*val_type* 参数用来给出一种日期类型，它可以是 DATE、TIME、DATETIME 或 TIMESTAMP。*format_type* 参数表明应该返回哪种风格的格式字符串，它可以是'EUR'（欧洲）、'INTERNAL'（MySQL 的内部表示形式）、'ISO'（ISO 9075。注意，不是 ISO 8601）、'JIS'（日本工业标准）或'USA'（美国）。

  GET_FORMAT()函数按照表 C-6 为每种 *val_type* 和 *format_type* 参数值的组合返回一个符合要求的格式字符串。

表 C-6

val_type	**format_type**	**格式字符串**
DATE	'EUR'	'%d.%m.%Y'
DATE	'INTERNAL'	'%Y%m%d'
DATE	'ISO'	'%Y-%m-%d'
DATE	'JIS'	'%Y-%m-%d'
DATE	'USA'	'%m.%d.%Y'
TIME	'EUR'	'%H.%i.%s'
TIME	'INTERNAL'	'%H%i%s'
TIME	'ISO'	'%H:%i:%s'
TIME	'JIS'	'%H:%i:%s'
TIME	'USA'	'%h:%i:%s %p'
DATETIME	'EUR'	'%Y-%m-%d %H.%i.%s'
DATETIME	'INTERNAL'	'%Y%m%d%H%i%s'
DATETIME	'ISO'	'%Y-%m-%d %H:%i:%s'
DATETIME	'JIS'	'%Y-%m-%d %H:%i:%s'
DATETIME	'USA'	'%Y-%m-%d %H.%i.%s'

  请注意，适用于 DATETIME 类型的'EUR'和'USA'格式字符串里的日期部分，和适用于 DATE 类型的'EUR'和'USA'格式字符串里的日期部分是不一样的。

- HOUR (*time*)

  返回时间值 *time* 里的小时数，其取值范围是 0 到 23。

```
HOUR('12:31:58')                                  → 12
HOUR(123158)                                      → 12
```

- LAST_DAY (*date*)

  返回由参数 *date* 所给定的月份里的最后一天的日期。参数 *date* 应该是一个日期值或一个日期/时间值。

```
LAST_DAY('2003-07-01')                          → '2003-07-31'
LAST_DAY('2003-07-01 12:30:00')                 → '2003-07-31'
```

- LOCALTIME ()

  LOCALTIMESTAMP ()

  这两个函数是 NOW()函数的同义词。括号是可选的。

- MAKEDATE (*year*, *day_of_year*)

  根据给定的年份和天数返回一个日期值。如果 *day_of_year* 参数的值小于 1，返回结果将是 NULL。

```
MAKEDATE(2010,365)                              → '2010-12-31'
MAKEDATE(2010,367)                              → '2011-01-02'
MAKEDATE(2010,0)                                → NULL
```

- MAKETIME (*hour*, *minute*, *second*)

  用给定的小时、分钟和秒数构造并返回一个时间值。如果有某个参数超出其可取值范围，则返回 *NULL*。分钟和秒数必须在 0 到 59 的范围内，小时数却允许超出此范围。如果小时数是一个负值，返回结果也将为负。

```
MAKETIME(0,0,0)                                 → '00:00:00'
MAKETIME(12,59,59)                              → '12:59:59'
MAKETIME(12,59,60)                              → NULL
MAKETIME(-12,59,59)                             → '-12:59:59'
```

- MICROSECOND (*expr*)

  返回给定的时间值或日期/时间值里的微秒数部分，该返回值的取值范围是 0 到 999999。

```
MICROSECOND('00:00:00.000001');                 → 1
MICROSECOND('2004-06-30: 23:59:59.5');          → 500000
```

- MINUTE(*time*)

  返回值：时间值 *time* 中的分钟数值，范围是 0 到 59。

```
MINUTE('12:31:58')                              → 31
MINUTE(123158)                                  → 31
```

- MONTH(*date*)

  返回值：日期值 *date* 中的月份值，范围是 0 到 12（0 表示日期部分没有月份表示）。

```
MONTH('2002-12-01')                             → 12
MONTH(20021201)                                 → 12
MONTH('2002-00-01')                             → 0
```

- MONTHNAME(*date*)

  返回值：日期值 *date* 中的月份名称（字符串格式），没有月份部分的日期为 NULL。

```
MONTHNAME('2002-12-01')                         → 'December'
MONTHNAME(20021201)                             → 'December'
MONTHNAME('2002-00-01')                         → NULL
```

- NOW ()

  以 *'CCYY-MM-DD hh:mm:ss'* 格式的 DATETIME 值返回当前地理时区的当前日期和时间。

```
NOW()                                           → '2008-05-01 18:36:09'
```

NOW()函数返回的是它所在的语句刚开始执行的那一刻的日期/时间值，不管那条语句会执行多长时间。如果是包含在某个存储例程或触发器里的 NOW()函数，它返回的是该例程或触发器刚开始执行的那一刻的日期/时间值。（建议把这种行为与 SYSDATE()函数作个比较。）

- PERIOD_ADD(*period*, *n*)

  返回值：时间段值 *period* 加上 *n* 个月后得到的结果。返回值的格式是 *CCYYMM*。输入参数 *period* 的格式可以是 *CCYYMM* 或 *YYMM*。需要特别注意的是：这两种格式都不是 MySQL 惯用的日期格式。

```
PERIOD_ADD(201002,12)                        → 201102
PERIOD_ADD(0802,-3)                          → 200711
```

- PERIOD_DIFF(*period1*, *period2*)

  返回值：时间段参数值相减得到的差，即它们之间相隔的月份数。这两个输入参数的格式是 *CCYYMM* 或 *YYMM*。需要特别注意的是：这两种格式都不是 MySQL 惯用的日期格式。

```
PERIOD_DIFF(200302,200202)                   → 12
PERIOD_DIFF(200711,0802)                     → -3
```

- QUARTER(*date*)

  返回值：日期值 *date* 确定的日子所在的季节值，范围是 1 到 4。

```
QUARTER('2008-12-01')                        → 4
QUARTER('2009-01-01')                        → 1
```

- SECOND(*time*)

  返回值：时间值 *time* 中的秒值，范围是 0 到 59。

```
SECOND('12:31:58')                           → 58
SECOND(123158)                               → 58
```

- SEC_TO_TIME(*second*)

  返回值：把秒数 *second* 转换为相应的时间值，格式为'*hh*:*mm*:*ss*'。

```
SEC_TO_TIME(29834)                           → '08:17:14'
```

- STR_TO_DATE (*str*, *format_str*)

  以 *format_str* 参数为格式字符串对参数 *str* 进行解释，并根据 *format_str* 参数里的限定符返回一个 TIME、DATE 或 DATETIME 值。可以利用这个函数来解释非 ISO 格式的日期/时间值。STR_TO_DATE()函数是 DATE_FAORMAT()函数的逆向操作，可以在 DATE_FAORMAT()函数里使用的所有格式限定符也都可以在 STR_TO_DATE()函数里合法使用。如果 *str* 参数值非法或是无法用给定的格式字符串解释，这返回结果将是 NULL。

```
STR_TO_DATE('3/16/1960','%m/%d/%Y')          → '1960-03-16'
STR_TO_DATE('12.20.32','%H.%i.%s')           → '12:20:32'
STR_TO_DATE('3/16/1960 12:20:32','%m/%d/%Y %H:%i:%s')
                                             → '1960-03-16 12:20:32'
STR_TO_DATE('3/16/1960','%m-%d-%Y')          → NULL
```

- SUBDATE(*date*,*INTERVAL interval*)

  SUBDATE(*date*,*expr*)

  对于第一种语法，SUBDATE()函数通过 *date* 参数读入一个日期值或日期/时间值，用它减去一

段时间间隔，最后返回计算的结果。它是 DATE_SUB()函数的同义词：

```
SUBDATE('2009-12-01',INTERVAL 1 MONTH)          → '2009-11-01'
```

对于第二种语法，SUBDATE()函数通过 *date* 参数读入一个日期值或日期/时间值，用它减去一个天数，最后返回计算的结果。这与 ADDDATE()函数的相应语法很相似。

```
SUBDATE('2009-12-01',30)                        → '2009-11-01'
```

- SUBTIME (*expr1*, *expr2*)

  用第一个表达式减去第二个表达式并返回其结果。*expr1* 应该是一个时间值或日期/时间值，*expr2* 应该是一个时间值。两个值都可以包含一个微秒部分。

```
SUBTIME('06:30:00.5','12:30:00.5')              → '-06:00:00.000000'
SUBTIME('2009-01-01 00:00:00','12:30:00')       → '2008-12-31 11:30:00'
```

- SYSDATE ()

  按'*CCYY-MM-DD hh*:*mm*:*ss*'格式返回当前地理时区的当前日期和时间 DATETIME。这个函数的行为类似于 NOW()函数，但从 MySQL 5.0.13 版开始，SYSDATE()函数返回的是它本身被调用的那一刻的日期/时间值，而 NOW()函数返回的是它所在的语句刚开始执行的那一刻的日期/时间值。(请参阅前面对 NOW()函数的描述。)如果想让 SYSDATE()函数的行为和 NOW()函数的一样，需要使用--sysdate-is-now 选项来启动服务器（从 MySQL 5.0.20 版开始）。

- TIME (*expr*)

  返回 *expr* 参数中的时间部分，*expr* 应该是一个时间值或一个日期/时间值。

```
TIME('16:15:00')                                → '16:15:00'
TIME('2005-03-12 16:15:00')                     → '16:15:00'
```

- TIME_FORMAT (*time*, *format*)

  根据 *format* 参数给定的格式字符串对 time 参数的值进行格式化并返回结果字符串。这个函数也接受 DATETIME 或 TIMESTAMP 类型的参数。这里使用的格式字符串和 DATE_FORMAT()函数里使用的相似，只是仅允许使用与时间有关的限定符；其他的限定符将导致一个 NULL 值或 0。

```
TIME_FORMAT('12:31:58','%H %i')                 → '12 31'
TIME_FORMAT(123158,'%H %i')                     → '12 31'
```

- TIME_TO_SEC (*time*)

  把参数 *time* 给出的时间值换算为相应的秒数。这个函数的返回值可以传递到 SEC_TO_TIME()函数，重新转换回一个时间值。

```
TIME_TO_SEC('08:17:14')                         → 29834
SEC_TO_TIME(29834)                              → '08:17:14'
```

  如果 *time* 参数给出的是一个 DATETIME 或 TIMESTAMP 值，TIME_TO_SECOND()函数将忽略日期部分。

```
TIME_TO_SEC('2012-03-26 08:17:14')              → 29834
```

- TIMEDIFF (*expr1*, *expr2*)

  计算并返回两个表达式的时间距离。第一个和第二个表达式分别是开始和结束时间。它们应该都是时间值或者都是日期/时间值，时间值和日期/时间值不能用。

```
TIMEDIFF('00:00:00','09:30:45')                 → '-09:30:45'
TIMEDIFF('09:30:45','00:00:00')                 → '09:30:45'
```

- TIMESTAMP (*expr1*[, *expr2*])

  如果只有一个参数，这个函数将把 *expr1* 参数给定的日期值或日期/时间值转换为一个 TIMESTAMP 值。如果有两个参数，这个函数将把参数 *expr2* 给出的时间值加到 *expr1* 参数值上，并返回计算结果 TIMESTAMP 值。

```
TIMESTAMP('1985-12-14');                        → '1985-12-14 00:00:00'
TIMESTAMP('1985-12-14 09:00:00');               → '1985-12-14 09:00:00'
TIMESTAMP('1985-12-14','18:00:00');             → '1985-12-14 18:00:00'
TIMESTAMP('1985-12-14 09:00:00','18:00:00');    → '1985-12-15 03:00:00'
TIMESTAMP('1985-12-14 09:00:00','-18:00:00');   → '1985-12-13 15:00:00'
```

- TIMESTAMPADD (*interval*, *expr1*, *expr2*)

  把 *expr1* 解释为一个以 *interval* 参数值为单位的整数，把该整数与 *expr2* 参数给出的日期值或日期/时间值相加，最后返回计算结果。允许使用的 *interval* 参数值是 FRAC_SECOND、SECOND、MINUTE、HOUR、DAY、WEEK、MONTH、QUARTER 和 YEAR，这些值在使用时都可以加上一个 SQL_TSI_前缀。从 MySQL 5.0.60/5.1.24 版开始，FRAC_SECOND 关键字已逐渐被淘汰，代替它来指定以微秒为计时单位的关键字是 MICROSECOND。

```
TIMESTAMPADD(DAY,12,'1995-07-01')               → '1995-07-13'
TIMESTAMPADD(MONTH,12,'1995-07-01')             → '1996-07-01'
TIMESTAMPADD(SQL_TSI_MONTH,12,'1995-07-01')     → '1996-07-01'
```

- TIMESTAMPDIFF (*interval*, *expr1*, *expr2*)

  计算 *expr1* 和 *expr2* 参数所给出的日期值或日期/时间值之间的差距，然后以 *interval* 参数值为单位返回计算结果。这里允许使用的 *interval* 参数值和 TIMESTAMPADD()函数里允许使用的一样。

```
TIMESTAMPDIFF(DAY,'1995-07-01','1995-08-01')    → 31
TIMESTAMPDIFF(MONTH,'1995-07-01','1995-08-01')  → 1
```

- TO_DAYS (*date*)

  把 *date* 参数所给出的日期值转换为从公元零年开始计算的天数并返回结果数值。这个函数的返回值可以传递到 FROM_DAYS()，重新转换回一个日期值。

```
TO_DAYS('2010-12-01')                           → 734472
FROM_DAYS(734472 - 365)                         → '2009-12-01'
```

  如果 *date* 参数给出的是一个 DATETIME 或 TIMESTAMP 值，TO_DAYS()函数将忽略时间部分。

```
TO_DAYS('2010-12-01 12:14:37')                  → 734472
```

  TO_DAYS()函数的设计覆盖范围仅限于公历出现以来的日期（1582 年以后）。

- UNIX_TIMESTAMP ()

  UNIX_TIMESTAMP (*date*)

  如果不带任何参数，这个函数将返回从 UTC 基准时间'1970-01-01 00:00:00'开始计算的秒数。如果通过 *date* 参数给出了一个日期值，它将返回 *date* 参数日期与基准时间之间的秒数。*date* 参数可以用多种方法给出：以 DATE、DATETIME 或 TIMESTAMP 形式给出，或者以 CCYYMMDD 或 YYMMDD 格式的数值形式给出。服务器将把 *date* 参数值解释为一个当前时区中的值并转换

为一个 UTC 格式的值，但来自某个 TIMESTAMP 数据列的值例外（因为存放在 TIMESTAMP 数据列里的值已经是 UTC 时间的格式了）。

```
UNIX_TIMESTAMP()                          → 1209685069
UNIX_TIMESTAMP('2007-12-01')              → 1196488800
UNIX_TIMESTAMP(20071201)                  → 1196488800
```

- UTC_DATE ()

把当前 UTC 日期值 DATE 按'*CCYY-MM-DD*'格式返回，其中的括号部分可以省略。

```
UTC_DATE()                                → '2008-05-01'
```

- UTC_TIME ()

把当前 UTC 时间值 TIME 按'*hh*:*mm*:*ss*'格式返回，其中的括号部分可以省略。

```
UTC_TIME()                                → '23:37:56'
```

- UTC_TIMESTAMP ()

按'*CCYY-MM-DD hh*:mm:*ss*'格式返回当前 UTC 日期和时间值 DATETIME 其中的括号部分可以省略。

```
UTC_TIMESTAMP()                           → '2008-05-01 23:38:02'
```

- WEEK (*date*[, *mode*])

如果只向这个函数传递了一个 *date* 参数，它将返回一个 0 到 53 之间的整数值，表明由 *date* 参数所给定的日期落在该年的第几个星期里——此时以星期日作为每个星期的第一天。如果向这个函数传递了两个参数，它将返回一个有着同样含义的整数值，但要根据 *mode* 参数的值来决定以星期几为每个星期的第一天，返回值的取值范围是从 0 到 53 还是从 1 到 53。表 C-7 列出了所有允许使用的 *mode* 参数值及其含义。

表　C-7

Mode 参数值	起始日	返回值的范围	该年的第一个星期
0	星期日	0..53	第一个包含星期日的星期
1	星期一	0..53	第一个多于 3 天的星期
2	星期日	1..53	第一个包含星期日的星期
3	星期一	1..53	第一个多于 3 天的星期
4	星期日	0..53	第一个多于 3 天的星期
5	星期一	0..53	第一个包含星期一的星期
6	星期日	1..53	第一个多于 3 天的星期
7	星期一	1..53	第一个包含星期一的星期

如果 *mode* 参数值缺失，这个函数将根据 default_week_format 系统变量的值来决定上述条件。

```
WEEK('2003-12-08')                        → 49
WEEK('2003-12-08',0)                      → 49
WEEK('2003-12-08',1)                      → 50
```

如果 WEEK()函数的返回值是 0，则表明 *date* 参数所给出的日期早于该年的第一个星期起始日（星期日或星期一，视 *mode* 参数值而定）。

```
WEEK('2005-01-01')                          → 0
DAYNAME('2005-01-01')                       → 'Saturday'
WEEK('2006-01-01',1)                        → 0
DAYNAME('2006-01-01')                       → 'Sunday'
```

- WEEKDAY (*date*)

  返回一个整数值以表明由 *date* 参数所给定的日期是星期几，如果无法确定，则返回 NULL。返回值的取值范围从 0(对应于星期一)到 6(对应于星期日)。相关信息参见关于 DAYOFWEEK() 函数的说明。

```
WEEKDAY('2002-12-08')                       → 6
DAYNAME('2002-12-08')                       → 'Sunday'
WEEKDAY('2002-12-16')                       → 0
DAYNAME('2002-12-16')                       → 'Monday'
WEEKDAY('2002-12-00')                       → NULL
```

- WEEKOFYEAR (*date*)

  这个函数等效于 WEEK(*date*, 3)。

- YEAR (*date*)

  返回一个整数值以表明由 *date* 参数所给定的日期落在哪一年里。

```
YEAR('2002-12-01')                          → 2002
YEAR(20021201)                              → 2002
```

- YEARWEEK (*date*[, *mode*])

  返回一个 *CCYYMM* 格式的整数值以表明由 *date* 参数所给定的日期落在该年的第几个星期里。如果还给出了 *mode* 参数，其含义与 WEEK() 函数里的相同。

```
YEARWEEK('2006-01-01')                      → 200601
YEARWEEK('2006-01-01',0)                    → 200601
YEARWEEK('2006-01-01',1)                    → 200552
```

  请注意，如果 *date* 参数值所给定的日期落在某年的第一个星期或最后一个星期里，YEARWEEK() 函数的返回值里的年份数字可能与 *date* 参数值里的不一样。

```
WEEK('2008-01-01')                          → 0
YEARWEEK('2008-01-01')                      → 200752
```

C.2.6 汇总函数

汇总函数的返回值是根据一组值计算出来的一个结果。这个结果是根据查询结果中的非 NULL 值统计出来的（只有 COUNT(*) 函数是个例外，它将统计所有的数据行）。汇总函数既可以对整个查询结果集进行统计，也可以对查询结果按某种规则进行归组（比如查询语句里使用了 GROUP BY 子句的情况）后得到的各个子集进行统计。有关这方面的详细讨论见 1.4.9 节的第 9 小节。

这一小节里的示例要用到一个下面这样的 mytbl 数据表：它有一个整数数据列 mycol，各数据行在这个数据列的值依次是 1、3、5、5、7、9、9 和 NULL，如下所示：

```
mysql> SELECT mycol FROM mytbl;
+-------+
| mycol |
+-------+
```

```
|     1 |
|     3 |
|     5 |
|     5 |
|     7 |
|     9 |
|     9 |
|  NULL |
+-------+
```

- AVG([DISTINCT] *expr*)

  返回值：表达式 *expr* 计算结果的平均值；涉及查询结果集里的全体非 NULL 值。如果没有非 NULL 值，则返回 NULL。

```
SELECT AVG(mycol) FROM mytbl                    → 5.5714
SELECT AVG(mycol)*2 FROM mytbl                  → 11.1429
SELECT AVG(mycol*2) FROM mytbl                  → 11.1429
```

  自 MySQL 5.0.3 起开始允许有 DISTINCT，它将令 AVG()返回各个 *expr* 值的平均值。

- BIT_AND(*expr*)

  返回值：表达式 *expr* 计算结果的按位“与”操作结果，涉及查询结果集里的全体非 NULL 值。如果都为 NULL，则返回~0。

```
SELECT BIT_AND(mycol) FROM mytbl                → 1
```

- BIT_OR(*expr*)

  返回值：表达式 *expr* 计算结果的按位“与”操作结果，涉及查询结果集里的全体非 NULL 值。如果都为 NULL，则返回 0。

```
SELECT BIT_OR(mycol) FROM mytbl                 → 15
```

- BIT_XOR(*expr*)

  返回涉及选定行中所有非 NULL 值的 *expr* 的按位异或值。如果都为 NULL 值，则返回 0。

```
SELECT BIT_XOR(mycol) FROM mytbl                → 5
```

- COUNT(*expr*)

  COUNT(*)

  COUNT(DISTINCT *expr1*,*expr2*,...)

  有一个表达式参数时，返回查询结果集里的非 NULL 值的个数。如果都为 NULL 值，返回 0。COUNT(*)将返回查询结果集里全体数据行（不管它们是不是 NULL 值）的个数，如下所示：

```
SELECT COUNT(mycol) FROM mytbl                  → 7
SELECT COUNT(*) FROM mytbl                      → 8
```

  对于 MyISAM 数据表，不带 WHERE 子句的 COUNT(*)被优化成直接返回 FROM 子句所指定的数据表里的数据行总数；如果 FROM 子句指定了多个数据表，COUNT(*)将返回各数据表的数据行总数的乘积，如下所示：

```
SELECT COUNT(*) FROM mytbl AS m1 INNER JOIN mytbl AS m2
                                                → 64
```

  还可以用 COUNT(DISTINCT)统计出查询结果里有多少种各不相同的非 NULL 值，如下所示：

```
SELECT COUNT(DISTINCT mycol) FROM mytbl         → 5
```

```
SELECT COUNT(DISTINCT MOD(mycol,3)) FROM mytbl    → 3
```

如果给出了多个表达式，COUNT(DISTINCT)的返回值将是全体表达式非 NULL 值计算结果的各不相同的组合的总数。

- GROUP_CONCAT ([DISTINCT] *val_list* [ORDER BY ...] [SEPARATOR *str*])

  这个函数把字符串列表 *val_list* 中的所有非 NULL 值合并在一起并返回其结果。如果 val_list 列表中没有非 NULL 值，则返回 NULL。DISTINCT 关键字可以用来剔除重复出现的字符串，ORDER BY 用来对结果进行排序，SEPARATOR 用来指定字符串之间的分隔符。在默认的情况下，这个函数不剔除重复值、不对结果排序，也不使用逗号作为分隔符。

  GROUP_CONCAT()函数的返回值的最大长度受限于 group_concat_max_len 系统变量的值。你可以改变这个变量的值以获得更长的合并结果。

```
mysql> CREATE TABLE t (name CHAR(10));
mysql> INSERT INTO t VALUES('dog'),('cat'),('rat'),('dog'),('rat');
mysql> SELECT GROUP_CONCAT(name) FROM t;
+---------------------+
| GROUP_CONCAT(name)  |
+---------------------+
| dog,cat,rat,dog,rat |
+---------------------+
mysql> SELECT GROUP_CONCAT(name SEPARATOR ':') FROM t;
+----------------------------------+
| GROUP_CONCAT(name SEPARATOR ':') |
+----------------------------------+
| dog:cat:rat:dog:rat              |
+----------------------------------+
mysql> SELECT GROUP_CONCAT(name ORDER BY name DESC) FROM t;
+---------------------------------------+
| GROUP_CONCAT(name ORDER BY name DESC) |
+---------------------------------------+
| rat,rat,dog,dog,cat                   |
+---------------------------------------+
mysql> SELECT GROUP_CONCAT(DISTINCT name ORDER BY name) FROM t;
+-------------------------------------------+
| GROUP_CONCAT(DISTINCT name ORDER BY name) |
+-------------------------------------------+
| cat,dog,rat                               |
+-------------------------------------------+
```

- MAX ([DISTINCT] *expr*)

  根据表达式 *expr* 对被选中的数据行里的所有非 NULL 值进行计算，并返回计算结果中的最大值。如果没有非 NULL 值，则返回 NULL。MAX()函数还可以用于字符串值或日期/时间值，此时它将返回字符串意义或日期/时间意义上的最大值。

```
SELECT MAX(mycol) FROM mytbl                    → 9
```

  DISTINCT 关键字将导致 MAX()函数在筛选最大值时剔除重复的 *expr* 值（这并不会改变返回结果）。

- MIN ([DISTINCT] *expr*)

  根据表达式 *expr* 对被选中的数据行里的所有非 NULL 值进行计算，并返回计算结果中的最小

值。如果没有非 NULL 值，则返回 NULL。MIN()函数还可以用于字符串值或日期/时间值，此时它将返回字符串意义或日期/时间意义上的最小值。

```
SELECT MIN(mycol) FROM mytbl                    → 1
```

DISTINCT 关键字将导致 MIN()函数在筛选最小值时剔除重复的 expr 值(这并不会改变返回结果)。

- STD(*expr*)

  STDDEV(*expr*)

  STDDEV_POP(*expr*)

  根据表达式 *expr* 对被选中的数据行里的所有非 NULL 值进行计算，并返回计算结果的样本标准偏差（population standard deviation)。如果没有非 NULL 值，则返回 NULL。

  ```
  SELECT STDDEV_POP(mycol) FROM mytbl             → 2.7701
  ```

  STDDEV_POP()函数是从 MySQL 5.0.3 版开始引入的。

- STDDEV_SAMP (*expr*)

  根据表达式 *expr* 对被选中的数据行里的所有非 NULL 值进行计算，并返回计算结果的样本标准偏差（sample standard deviation)。如果没有非 NULL 值，则返回 NULL。

  ```
  SELECT STDDEV_SAMP(mycol) FROM mytbl            → 2.9921
  ```

  STDDEV_SAMP()函数是从 MySQL 5.0.3 版开始引入的。

- SUM ([DISTINCT] *expr*)

  根据表达式 *expr* 对被选中的数据行里的所有非 NULL 值进行计算,并返回计算结果的累加值。如果没有非 NULL 值，则返回 NULL。

  ```
  SELECT SUM(mycol) FROM mytbl                    → 39
  ```

  DISTINCT 关键字将导致 SUM()函数在计算累加值时剔除重复的 expr 值。

- VARIANCE (*expr*)

  VAR_POP (*expr*)

  根据表达式 *expr* 对被选中的数据行里的所有非 NULL 值进行计算，并返回计算结果的总方差(population variance)。如果没有非 NULL 值，则返回 NULL。

  ```
  SELECT VAR_POP(mycol) FROM mytbl                → 7.6735
  ```

  VAR_POP()函数是从 MySQL 5.0.3 版开始引入的。

- VAR_SAMP (*expr*)

  根据表达式 *expr* 对被选中的数据行里的所有非 NULL 值进行计算，并返回计算结果的样本方差（sample variance)。如果没有非 NULL 值，则返回 NULL。

  ```
  SELECT VAR_SAMP(mycol) FROM mytbl               → 8.9524
  ```

  VAR_SAMP()函数是从 MySQL 5.0.3 版开始引入的。

C.2.7　数据加密和压缩函数

这些函数用来完成各种与数据安全有关的操作，如字符串的加密或压缩。在这类函数中，有一些是成对出现的，其中一个用来进行加密，另一个则用来解密。这些成对出现的函数通常都要使用同一

个字符串来充当密钥或口令。要想在解密后得到原先的数据，就必须使用在加密它们时使用的同一个密钥来进行解密；否则，解密结果将毫无价值。

加密函数的返回值通常都是一些二进制字符串。因此，如果你打算把它们保存到数据库里，就应该使用一种属于 BLOB 系列的数据列类型。

- AES_DECRYPT(*str*, *key_str*)

  对于已加密的通过调用 AES_ENCRYPT()获得的字符串，使用密钥字符串 *key_str* 进行解密，返回结果字符串。如果输入参数中有 NULL 值，则返回 NULL。

```
AES_DECRYPT(AES_ENCRYPT('secret','scramble'),'scramble')
                                                → 'secret'
```

- AES_ENCRYPT(*str*, *key_str*)

  采用 AES（Advanced Encryption Standard）算法以密钥 *key_str* 对字符串 *str* 进行加密而得到一个二进制字符串，它使用的密钥长度是 128 位。如果输入参数中有 NULL 值，则返回 NULL。本函数的加密结果可以用 AES_DECRYPT()函数和同一个密钥解密为原先的字符串。

- COMPRESS（*str*）

  把参数字符串的压缩版本按二进制字符串格式返回，如果没有用压缩库编译服务器，则返回 NULL。

- DECODE(*str*, *key_str*)

  返回值：用密钥 *key_str* 对调用 ENCODE()后得到的加密字符串 *str* 进行解密而得到的结果字符串。如果字符串 *str* 是 NULL 值，则返回 NULL。

```
DECODE(ENCODE('secret','scramble'),'scramble')    → 'secret'
```

- DES_DECRYPT(*str*[,*key_str*]

  对用 DES_ENCRYPT()函数进行加密而得到的二进制字符串 *str* 进行解密。如果没有启用系统上的 SSL 支持功能或者解密操作失败，DES_DECRYPT()将返回 NULL。

  如果给出了 *key_str* 参数，DES_DECRYPT()函数就将把它用作解密密钥。如果没有给出 *key_str* 参数，DES_DECRYPT()函数就将使用一个来自服务器 DES 密钥文件里的密钥来对字符串 *str* 进行解密。这个密钥的编号将由加密字符串第一个字节里的第 0 到第 6 比特决定，而密钥文件在服务器里的存放地点则是由你（或别人）在启动服务器时用--des-key-file 选项指定的。如果加密和解密时使用的密钥不一致，就会导致毫无意义的结果。

  如果字符串 *str* 看起来不像是经加密而得到的结果，比如字符串 str 第一个字节的第 7 位没有被置位（即等于 0）时，DES_DECRYPT()函数将原样返回字符串 str。

  只有拥有 SUPER 权限的用户才能使用只带一个参数的 DES_DECRYPT()函数。

- DES_ENCRYPT(*str* [, {*key_num*|*key_str*}])

  返回值：采用 DES 算法对字符串 *str* 进行加密而得到的一个二进制字符串。本函数的加密结果可以用 DES_DECRYPT()函数解密。如果系统上的 SSL 支持功能没有被启用或者加密操作失败，DES_ENCRYPT()将返回 NULL。

  如果给出了 *key_str* 参数，DES_ENCRYPT()函数就将把它用作加密密钥。如果给出了 *key_num* 参数（它应该是一个 0 到 9 之间的整数），DES_DECRYPT()函数就将使用服务器 DES 密钥文件里编号为 key_num 的那个密钥来对字符串 *str* 进行加密。如果 *key_str* 和 *key_num* 参数都

没有给出，就将使用 DES 密钥文件里的第一个密钥（注意：它与你把 *key_num* 参数设置为 0 时使用的密钥不见得是同一个）来进行加密。

结果字符串的第一个字节能让我们知道加密工作是如何进行的。这个字节的第 7 位应该被置位为 1，而第 0 到第 6 位则构成了一个密钥编号：如果这个编号是一个 0 到 9 之间的数字，就说明加密工作是用服务器 DES 密钥文件里对应于该编号的那个密钥来完成的。如果这个编号等于 127，就说明加密工作是用一个 *key_str* 参数来完成的。比如说，如果加密时使用的密钥编号为 3，加密结果字符串第一个字节的值就应该是 131（即 128+3）。如果加密时使用的是一个 *key_str* 参数，加密结果字符串第一个字节的值就应该是 255（即 128+127）。

对于使用 *key_num* 参数的加密操作，服务器将从某个 DES 密钥文件里读出相应的密钥字符串，而这个 DES 密钥文件在服务器里的存放地点则是由你（或别人）在启动服务器时用 `--des-key-file` 选项指定的。密钥在密钥文件里的存放格式如下所示：

key_num key_str

每个 *key_num* 都是一个 0 到 9 之间的整数，而 *key_st* 则是与之对应的加密密钥；*key_num* 与 *key*_st 之间要用至少一个空白符加以分隔。密钥文件里的各行允许按任意先后顺序出现。

`DES_ENCRYPT()`函数并不要求用户必须拥有 `SUPER` 权限才能使用来自 DES 密钥文件里的密钥，这与 `DES_DECRYPT()`函数是不同的。（谁都可以使用 DES 密钥文件里的密钥去加密自己的信息，但只有那些有足够权限的用户才能解密。）

- `ENCODE(`*str*`,` *key_str*`)`

  返回值：用密钥字符串 *key_str* 对字符串 *str* 进行加密而得到的一个二进制字符串。本函数的加密结果可以用 `DECODE()`函数和同一个密钥解密为原先的字符串。

- `ENCRYPT(`*str*`[,` *salt*`])`

  返回值：字符串 *str* 的加密结果字符串。如果输入参数中有 `NULL` 值，则返回 `NULL`。这是一个不可逆的加密过程。*salt* 参数（如果给出的话）必须是一个由两个或更多字符组成的字符串。对于一个给定的 *salt* 值，不管你对字符串 *str* 进行多少次加密，其结果都将是一样的。如果不带 *salt* 参数，MySQL 使用一个随机值，同样的 `ENCRYPT()`函数调用对字符串 *str* 每次加密的结果就将是不同的。

```
ENCRYPT('secret','AB')                          → 'ABS5SGh1EL6bk'
ENCRYPT('secret','AB')                          → 'ABS5SGh1EL6bk'
ENCRYPT('secret')                               → '9ai/2GobGFmXY'
ENCRYPT('secret')                               → 'Ea5Y.zUlAoUz.'
```

  `ENCRYPT()`函数使用 Unix 操作系统的底层 `crypt()`系统调用来完成加密操作，所以针对 `crypt()`的各项规定也同样适用于它。比如说，在某些系统上，`crypt()`只对字符串的前 8 个字符加密。如果 `crypt()`在你的系统上不存在或不可用，`ENCRYPT()`函数的返回值将永远是 `NULL`。

  只要有可能，就不要让 *str* 包含多字节字符，除非它使用的是 `utf8` 字符集，这是因为 `crypt()` 函数总是把 `NULL` 字符视为字符串的结束标记。

- `MD5(`*str*`)`

  返回值：用 RSA 数据安全公司的 MD5 信息标记算法为字符串 *str* 生成的一个 128 位的校验和。这个返回值是一个由 32 个十六进制数字构成的二进制字符串；但如果字符串 *str* 是 `NULL`，

那么返回值也将是 NULL。

```
MD5('secret')            → '5ebe2294ecd0e0f08eab7690d2a6ee69'
```

请参阅关于 SHA1()函数的介绍。

- OLD_PASSWORD(*str*)
  返回加密口令值，这是 MySQL 4.1 之前的版本返回的。
- PASSWORD(*str*)
  返回值：根据字符串 *str* 而计算出来的一个加密字符串，其格式与 MySQL 在它存放用户口令的权限表里使用的格式完全一样。这是一个不可逆的加密过程。

```
PASSWORD('secret')       → '*14E65567ABDB5135D0CFD9A70B3032C179A49EE7'
```

  需要提醒大家的是，PASSWORD()函数所使用的算法与 Unix 用来加密账户口令的算法是不一样的。如果你想要的是后一种加密效果，就应该使用 ENCRYPT()函数。
  如果 old_passwords 系统变量的值不为零，PASSWORD()函数返回的将是一个使用 MySQL 4.1 版之前的散列算法而得到的口令。此时，PASSWORD()函数和 OLD_PASSWORD()函数将返回同样的值。可以通过发出 SET GLOBAL old_passwords = 1 语句或是在启动 MySQL 服务器时给出--old_passwords 选项来启用 old_passwords 系统变量。
- SHA1(*str*)
  SHA(*str*)
  返回值：用 SHA（Secure Hash Algorithm，安全散列算法）算法为字符串 *str* 生成的一个 160 位的校验和。这个返回值是一个由 40 个十六进制数字构成的二进制字符串；但如果字符串 *str* 是 NULL，那么返回值也将是 NULL。

```
SHA1('secret')           → 'e5e9fa1ba31ecd1ae84f75caaa474f3a663f05f4'
```

- SHA2(*str*,*hash_length*)
  这个函数和 SHA1()函数很相似，但安全性更高。它的第一个参数用来给出将被加密的字符串，第二个参数用来设定加密结果的长度是多少个二进制位。*hash_length* 参数的值只能是 224、256、384 或 512。加密结果是一个以十六进制数字表示的二进制字符串。如果两个参数至少有一个是 NULL，或者 *hash_length* 参数值是非法的，这个函数将返回 NULL。

```
SHA2('secret',224)
             → '95c7fbca92ac5083afda62a564a3d014fc3b72c9140e3cb99ea6bf12'
```

  SHA2()函数是从 MySQL 6.0.5 版开始引入的。
- UNCOMPRESS (*str*)
  给定一个用 COMPRESS()函数压缩而来的字符串，UNCOMPRESS()函数将返回其原始明文。如果参数不是一个压缩字符串，或是在编译 MySQL 服务器时根本没有把支持压缩功能的函数库包括进去，它将返回 NULL。
- UNCOMPRESSED_LENGTH (*str*)
  给定一个用 COMPRESS()函数压缩而来的字符串，这个函数将返回其原始明文的长度。如果在编译服务器时根本没有把支持压缩功能的函数库包括进去，它将返回 NULL。

C.2.8　命名锁函数

在这一节里，我们将对与命名锁（advisory locking，也叫做“通知锁”、“合作锁”等）机制有关的几个函数进行介绍。利用这些函数可以编写能够根据某个命名锁的状态而协调彼此操作的应用程序。在这一机制中，最主要的函数是用来申请命名锁的 GET_LOCK()函数和用来释放命名锁的 RELEASE_LOCK()函数。还有两个是用来查询命名锁状态的 IS_FREE_LOCK()函数和用来确定哪个客户正拥有着某给定命名锁的 IS_USED_LOCK()函数。

从本质上讲，命名锁只是你锁定的一个名字，而这个名字不过是一个字符串而已。命名锁是私有的，只能由当前拥有它的那个客户来释放；但也是公有的，任何一个客户都可以查询其状态。

GET_LOCK(*str*, *timeout*)函数用来获得命名锁，其中 *str* 参数是该命名锁的名字，*timeout* 参数是一段以秒为单位的倒计时时间。如果在倒计时结束前成功地获得了该命名锁，GET_LOCK()函数将返回 1；否则，它将返回 0；如果发生错误，它将返回 NULL。

timeout 参数值是尝试获得一个命名锁时的等待时间，而不是锁的持续时间。在获得命名锁之后，它在被释放之前将一直发挥效力。

下面的调用将尝试获得一个名为'Nellie'的命名锁，最多等待 10 秒的时间：

```
GET_LOCK('Nellie',10)
```

命名锁只作用于字符串名字本身。它不锁定数据库、数据表或数据表里的任何一个数据行或数据列。换句话说，命名锁不会阻止任何其他客户对数据库里的数据表做任何事情，所以 GET_LOCK()函数所实现的锁定机制隐含着“自愿遵守”的意思——它只是让彼此协调操作的客户来确定是否让命名锁生效。

已经获得命名锁的客户将阻止其他客户（对一个与 MySQL 服务器同时建立了多条连接的多线程客户而言，就是其他线程）锁定该名字。假设客户 A 已经锁定了字符串'Nellie'，当客户 B 试图锁定同一个字符串时，客户 B 将被阻断，直到客户 A 释放这个命名锁或是客户 B 的倒计时时间结束。如果客户 A 在客户 B 的倒计时时间结束前释放了这个命名锁，客户 B 的这次申请将成功；否则，客户 B 失败。

因为两个客户不能同时锁定同一个字符串，我们就可以在应用程序里把一个字符串用作某个命名锁的名字，根据该命名锁的状态去判断能否安全地进行与该名字相关的操作。比如说，我们可以给数据表里的数据行设置一个命名锁，让应用程序根据其状态去协调对该数据行的操作。

要想明确地释放某个命名锁，以该命名锁的名字为参数调用 RELEASE_LOCK()函数即可：

```
RELEASE_LOCK('Nellie')
```

如果该命名锁释放成功，RELEASE_LOCK()函数将返回 1；如果该命名锁正由另一条连接持有（你只能释放自己持有的命名锁），它将返回 0；如果该命名锁不存在，它将返回 NULL。

每个客户连接一次只能锁定一个字符串，所以当持有某个命名锁的客户发出另一个 GET_LOCK()调用时，它当前持有的命名锁将自动释放。旧命名锁的释放将发生在新命名锁的获得之前，即使新旧命名锁的名字完全一样也不例外。还有，当客户与服务器之间的连接断开时，它持有的命名锁也将自动释放。所以要特别注意：如果你执行了一个运行时间非常长的客户，并且很长时间没有与之互动，它持有的命名锁就有可能因为连接超时而被自动释放。

如果想测试某个命名锁的状态，有两种做法可供选择。

- 调用IS_FREE_LOCK(*str*)函数。如果*str*参数所给定的名字可用（即它目前还没被用作一个命

名锁），该函数将返回1；如果该命名锁正被其他客户使用，返回0；如果发生错误，返回NULL。

- ❑ 调用IS_USED_LOCK(*str*)函数。如果不存在命名锁，该函数将返回NULL；如果该命名锁正被其他客户使用，返回正持有该命名锁的那个客户的连接ID。

还可以使用 GET_LOCK(*str*, 0)调用立即测试 *str* 参数所给定的命名锁是否正被其他客户使用。不过，这个简便方法有这样一个副作用：如果那个字符串当前未被锁定，你发出的调用就会锁定之，而这意味着你必须适时发出 RELEASE_LOCK()调用以免影响其他客户。

如果用来给出命名锁名字的那个参数是 NULL，所有的命名锁函数都将返回 NULL。

- ❑ GET_LOCK(*str*, *timeout*)

  在 *timeout* 秒的时间内，以 *str* 字符串为名字申请一个命名锁。如果在倒计时结束前成功地获得了该命名锁，返回 1；否则，返回 0；如果发生错误，返回 NULL。
- ❑ IS_FREE_LOCK(*str*)

  测试以*str*参数值为名字的命名锁的状态。如果*str*参数所给定的名字可用（即它目前还没被用作命名锁），返回1；如果该命名锁正被其他客户使用，返回0；如果发生错误，返回NULL。
- ❑ IS_USED_LOCK(*str*)

  检查是否存在以*str*参数值为名字的命名锁。如果存在，IS_USED_LOCK()函数将返回创建该命名锁的客户的连接ID；否则，返回NULL。
- ❑ RELEASE_LOCK(*str*)

  释放以*str*参数值为名字的命名锁。如果释放成功，返回1；如果该命名锁正由另一条连接持有，返回0；如果该命名锁不存在，返回NULL。

C.2.9 空间函数

本节介绍与坐标值有关的函数，它们也被称为“几何函数”。关于空间数据类型的更多信息，请参阅第 3 章。

坐标值可以表示为 3 种格式：

- ❑ WKB（Well-Known Binary）格式
- ❑ WK（Well-Known Text）格式
- ❑ MySQL内部格式

在 MySQL 里，函数的坐标值参数要求格式正确。如果传递的坐标值不是上述格式之一或是传递了一个非坐标值，这些函数将返回 NULL。可以利用一些函数把坐标值转换为不同的格式。

绝大多数以坐标值为参数的函数都要求其参数是 MySQL 内部格式，而 MySQL 在把坐标值保存到有空间数据类型的数据列里时也只使用内部格式。可以使用其他数据类型（如 BLOB）的数据列来保存 TEXT 和 WKB 格式的值。

坐标值可以与一个坐标参照 ID（SRID，Spatial Reference ID）相关联。在 MySQL 里，许多空间函数都有一个可选的 SRID 参数。

说明 这些函数是遵照 OpenGIS 规范而实现的。在接下来的内容里，我们将说明 OpenGIS 规范里的哪些函数未在 MySQL 里实现，又有哪些函数的 MySQL 实现方式与 OpenGIS 规范里的描述有所不同。

1. **坐标值格式转换函数**

下列函数可以把 WKB 格式的坐标值转换为 MySQL 内部格式的坐标值。*wkb_expr* 参数代表某函数能够接受的某几何对象的 WKB 值，*srid* 参数代表可选的坐标参照标识符。

- GEOMCOLLFROMWKB(*wkb_expr*[,*srid*])
  GEOMETRYCOLLECTIONFROMWKB(*wkb_expr*[,*srid*])
  把 WKB 值转换为一个 GEOMETRYCOLLECTION 值。
- GEOMFROMWKB(*wkb_expr*[,*srid*])
  GEOMETRYFROMWKB(*wkb_expr*[,*srid*])
  把 WKB 值转换为一个 GEOMETRY 值。这个函数可以接受任意空间类型的 WKB 值。
- LINEFROMWKB(*wkb_expr*[,*srid*])
  LINESTRINGFROMWKB(*wkb_expr*[,*srid*])
  把 WKB 值转换为一个 LINESTRING 值。
- MLINEFROMWKB(*wkb_expr*[,*srid*])
  MULTILINESTRINGFROMWKB(*wkb_expr*[,*srid*])
  把 WKB 值转换为一个 MULTILINESTRING 值。
- MPOINTFROMWKB(*wkb_expr*[,*srid*])
  MULTIPOINTFROMWKB(*wkb_expr*[,*srid*])
  把 WKB 值转换为一个 MULTIPOINT 值。
- MPOLYFROMWKB(*wkb_expr*[,*srid*])
  MULTIPOLYGONFROMWKB(*wkb_expr*[,*srid*])
  把 WKB 值转换为一个 MULTIPOLYGON 值。
- POINTFROMWKB(*wkb_expr*[,*srid*])
  把 WKB 值转换为一个 POINT 值。
- POLYFROMWKB(*wkb_expr*[,*srid*])
  POLYGONFROMWKB(*wkb_expr*[,*srid*])
  把 WKB 值转换为一个 POLYGON 值。

未实现的函数。OpenGIS 规范里描述的可选函数 POLYFROMWKB()和 POLYFROMWKB()可以对 WKB 值进行相应的格式转换，但 MySQL 未实现这些函数。

下列函数可以把 WKT 格式的坐标值转换为 MySQL 内部格式的坐标值。*wkt_expr* 参数代表某函数能够接受的某几何对象的 WKT 值，*srid* 参数代表可选的坐标参照标识符。

- GEOMCOLLFROMTEXT(*wkt_expr*[,*srid*])
  GEOMETRYCOLLECTIONFROMTEXT(*wkt_expr*[,*srid*])
  把 WKT 值转换为一个 GEOMETRYCOLLECTION 值。
- GEOMFROMTEXT(*wkt_expr*[,*srid*])
  GEOMETRYFROMTEXT(*wkt_expr*[,*srid*])
  把 WKT 值转换为一个 GEOMETRY 值。这个函数可以接受任意空间类型的 WKT 值。
- LINEFROMTEXT(*wkt_expr*[,*srid*])
  LINESTRINGFROMTEXT(*wkt_expr*[,*srid*])
  把 WKT 值转换为一个 LINESTRING 值。

- MLINEFROMTEXT(*wkt_expr*[,*srid*])
  MULTILINESTRINGFROMTEXT(*wkt_expr*[,*srid*])
  把 WKT 值转换为一个 MULTILINESTRING 值。
- MPOINTFROMTEXT(*wkt_expr*[,*srid*])
  MULTIPOINTFROMTEXT(*wkt_expr*[,*srid*])
  把 WKT 值转换为一个 MULTIPOINT 值。
- MPOLYFROMTEXT(*wkt_expr*[,*srid*])
  MULTIPOLYGONFROMTEXT(*wkt_expr*[,*srid*])
  把 WKT 值转换为一个 MULTIPOLYGON 值。
- POINTFROMTEXT(*wkt_expr*[,*srid*])
  把 WKT 值转换为一个 POINT 值。
- POLYFROMTEXT(*wkt_expr*[,*srid*])
  POLYGONFROMTEXT(*wkt_expr*[,*srid*])
  把 WKT 值转换为一个 POLYGON 值。

未实现的函数。OpenGIS 规范里描述的可选函数 POLYFROMTEXT()和 POLYFROMTEXT()可以对 WKT 值进行相应的格式转换，但 MySQL 未实现这些函数。

下列函数可以把 MySQL 内部格式的坐标值转换为 WKB 格式的相应值：

- ASBINARY(*geom*)
- ASWKB(*geom*)

下列函数可以把 MySQL 内部格式的坐标值转换为 WKT 格式的相应值：

- ASTEXT(*geom*)
- ASWKT(*geom*)

2. 坐标属性函数

下列函数将返回 *geom* 参数所代表的几何对象的属性，*geom* 参数是一个 MySQL 内部格式的坐标值，可以是任意类型。

- DIMENSION (*geom*)
  返回给定几何对象的维数。维数值的含义如表 C-8 所示。
  例如，POINT 类型的几何对象的维数是 0，LINESTRING 类型的几何对象的维数是 1，POLYGON 类型的几何对象的维数是 2。

表 C-8

维 数	说 明
−1	空几何对象
0	没有长度或面积的几何对象
1	零长度和零面积的几何对象
2	面积不为零的几何对象

- ENVELOPE (*geom*)

返回一个 POLYGON 值来表示给定几何对象的最小包围矩形。

- GEOMETRYTYPE (*geom*)

  返回一个字符串来表示给定几何对象的空间数据类型。

  GEOMETRYTYPE(GEOMFROMTEXT('LINESTRING(1 1,2 2)')) → 'LINESTRING'

- SRID (*geom*)

  返回一个整数值来表示给定几何对象的坐标参照 ID。

未实现的函数。OpenGIS 规范还定义了几种通用坐标属性函数，但 MySQL 没有实现它们：BOUNDARY()、ISEMPTY()和 ISSIMPLE()。

下列函数返回 *pt* 的属性，*pt* 参数必须是 MySQL 内部格式的 POINT 坐标值。

- X (*pt*)

  返回一个双精度浮点数来表示给定坐标点的 X 坐标值。

- Y (*pt*)

  返回一个双精度浮点数来表示给定坐标点的 Y 坐标值。

下列函数返回 *ls* 的属性，*ls* 参数必须是一个 MySQL 内部格式的 LINESTRING 类型的坐标值。

- ENDPOINT (*ls*)

  返回一个 POINT 值来表示 *ls* 的终点（最后一个坐标点）。

- GLENGTH (*ls*)

  返回一个双精度浮点数来表示 *ls* 的长度。

- ISCLOSED (*ls*)

  如果 *ls* 是封闭的，返回 1；如果 *ls* 不是封闭的，返回 0；如果 *ls* 是 NULL，返回−1。封闭的 LINESTRING 的起点和终点相互重合。

- NUMPOINTS (*ls*)

  返回 *ls* 里的坐标点的个数。

- POINTN (*ls*, *n*)

  返回一个 POINT 值来表示 *ls* 里的第 *n* 个坐标点。坐标点从 1 开始编号。

- STARTPOINT (*ls*)

  返回一个 POINT 值来表示 *ls* 的起点（第一个坐标点）。

未实现的函数。OpenGIS 规范还定义了适用于 LINESTRING 类型的属性函数 ISRING()，但 MySQL 没有实现它。

下列函数返回 *msl* 的属性，*mls* 参数必须是一个 MySQL 内部格式的 MULTILINESTRING 类型的坐标值。

- GLENGTH (*mls*)

  返回一个双精度浮点数来表示 *mls* 的长度。MULTILINESTRING 值的长度等于其各成员 LINESTRING 值的长度的总和。

- ISCLOSED (*mls*)

  如果 *mls* 是封闭的，返回 1；如果 *mls* 不是封闭的，返回 0；如果 *mls* 是 NULL，返回−1。封闭的 MULITLINESTRING 值必须满足以下条件：它的每个成员 LINESTRING 值的起点和终点相互重合。

下列函数返回 *poly* 的属性，*poly* 参数必须是一个 MySQL 内部格式的 POLYGON 类型的坐标值。

- ❑ AREA（*poly*）
  返回一个双精度浮点数来表示 *poly* 的面积。
- ❑ EXTERIORRING（*poly*）
  返回一个 LINESTRING 值来表示 *poly* 的外围线。
- ❑ INTERIORRINGN（poly,*n*）
  返回一个 LINESTRING 值来表示 *poly* 中的第 *n* 个内围线。内围线从 1 开始编号。
- ❑ NUMINTERIORRINGS（*poly*）
  返回 *poly* 中的内围线的个数。

下列函数返回 *mpoly* 的属性，*mpoly* 参数必须是一个 MySQL 内部格式的 MULTIPOLYGON 类型的坐标值：

- ❑ AREA（*mpoly*）
  返回一个双精度浮点数来表示 *mpoly* 的面积。

未实现的函数。OpenGIS 规范还定义了适用于 MULTIPOLYGON 类型的属性函数 CENTROID()和 POINTONSURFACE()，但 MySQL 没有实现它们。

下列函数返回 *gc* 的属性，*gc* 参数必须是一个 MySQL 内部格式的 GEOMETRYCOLLECTION 类型的坐标值。

- ❑ GEOMETRYN(*gc*,*n*)
  返回 *gc* 中的第 *n* 个几何对象，返回值的类型取决于第 *n* 个几何对象的类型。几何对象从 1 开始编号。
- ❑ NUMGEOMETRIES(*gc*)
  返回 *gc* 中的几何对象的个数。

3. 坐标关系函数

MySQL 实现了以下函数，用于测试两个几何对象 *geom1* 和 *geom2* 之间的坐标关系。*geom1* 和 *geom2* 参数必须是 MySQL 内部坐标格式。这些函数是以每一个几何值的最小边界矩形（minimum bounding tectangle，MBR）为基础而比较的。

- ❑ MBRCONTAINS（*geom1*，*geom2*）
  如果 *geom1* 的最小边界矩形完全覆盖 *geom2* 的最小边界矩形，返回 1；否则，返回 0。
- ❑ MBRDISJOINT(*geom1*，*geom2*)
  如果 *geom1* 的最小边界矩形与 *geom2* 的不相邻，返回 1；否则，返回 0。如果两个几何对象没有任何交叉，它们就是不相邻的。
- ❑ MBREQUAL(*geom1*，*geom2*)
  如果 *geom1* 的最小边界矩形和 *geom2* 形状相同，返回 1；否则，返回 0。
- ❑ MBRINTERSECTS(*geom1*，*geom2*)
  如果 *geom1* 的最小边界矩形和 *geom2* 的最小边界矩形有交叉之处，返回 1；否则，返回 0。
- ❑ MBROVERLAPS(*geom1*，*geom2*)
  如果 geom1 的最小边界矩形和 *geom2* 的最小边界矩形相互重合，返回 1；否则，返回 0。
- ❑ MBRTOUCHES(*geom1*，*geom2*)
  如果 *geom1* 的最小边界矩形和 *geom2* 的最小边界矩形相切（邻接但没有任何重叠之处），返回 1；否则，返回 0。

- MBRWITHIN(*geom1*, *geom2*)

如果 *geom1* 的最小边界矩形被 *geom2* 的最小边界矩形完全覆盖，返回 1；否则，返回 0。

OpenGIS 规范定义了以下函数用于测试几何对象之间的坐标关系，它们目前在 MySQL 里的实现和相应的基于 MBR 的函数是一样的。

- CONTAINS(*geom1*, *geom2*)
- DISJOINT(*geom1*, *geom2*)
- EQUALS(*geom1*, *geom2*)
- INTERSECTS(*geom1*, *geom2*)
- OVERLAPS(*geom1*, *geom2*)
- TOUCHES(*geom1*, *geom2*)
- WITHIN (*geom1*, *geom2*)

未实现的函数。OpenGIS 规范还定义了以下几种坐标关系函数，但 MySQL 没有实现它们：CROSSES()、DISTANCE()和 RELATED()。

C.2.10　XML函数

本节将介绍两个函数，它们可以根据给定的 Xpath 表达式处理由 XML 代码片段构成的字符串，一个用来从 XML 代码片段中提取文本，另一个用来把 XML 代码片段中的匹配元素替换为另一个字符串后返回替换结果。

这些函数的 XML 字符串参数所包含的 XML 标记必须搭配正确，嵌套层次不得错乱。

这些函数的 Xpath 表达式参数必须符合 Xpath 1.0 规范。有关 Xpath 规范的基本信息可以在 http：//vwww.w3.org/TR/xpath 网站上查到。MySQL 对 Xpath 的支持并不完备，有关这方面的最新进展请查阅《MySQL 参考手册》。

- EXTRACTVALUE (*xml_str*, *xpath_expr*)

  根据 Xpath 表达式对 XML 字符串进行处理，返回该 Xpath 表达式在该 XML 字符串里匹配到的 XML 元素的第一个文本结点的内容。如果该 Xpath 表达式匹配到了多个 XML 元素的话，这个函数将以空格为分隔符把所有匹配元素的第一个文本结点合并在一起作为返回结果。

```
EXTRACTVALUE('<a><b>B</b><c>C</c></a>','//b')      → 'B'
EXTRACTVALUE('<a><b>B1</b><b>B2</b><b>B3</b></a>','//b')
                                                    → 'B1 B2 B3'
```

  如果没有找到任何匹配，这个函数将返回一个空字符串（这和匹配到一个 XML 元素、但该 XML 元素不包含任何文本内容的情况是一样的）。

  EXTRACTVALUE()函数是从 MySQL 5.1.5 版开始引入的。

- UPDATEXML (*xml_str*, *xpath_expr*, *xml_new*)

  根据 Xpath 表达式对 XML 字符串进行处理，把匹配到的 XML 元素替换为 *xml_new* 参数值并返回。如果没有找到任何匹配或是匹配到了多个 XML 元素，这个函数将返回该 XML 字符串本身而不对它做任何改动。

  UPDATEXML()函数是从 MySQL 5.1.5 版开始引入的。

C.2.11 杂项函数

MySQL 还有一些函数无法简单地划归到前面介绍的类别中。

- BENCHMARK (*n*, *expr*)

  对表达式 *expr* 重复求值 *n* 次。BENCHMARK()函数和其他函数不太一样，因为人们通常只在 mysql 客户程序里使用它。它的返回值永远是 0，没有任何实际的用途。真正让人感兴趣的是 mysql 程序在查询结果的下方显示的执行时间：

```
mysql> SELECT BENCHMARK(1000000,PASSWORD('secret'));
+--------------------------------------+
| BENCHMARK(1000000,PASSWORD('secret')) |
+--------------------------------------+
|                                     0 |
+--------------------------------------+
1 row in set (2.35 sec)
```

  这个时间只能大致评估 MySQL 服务器对给定表达式的求值操作到底有多快，因为它是客户端的现实时间，不是服务器端的 CPU 时间。这个时间可能受到多种因素的影响，如服务器端的负载情况、BENCHMARK()查询到达时 MySQL 服务器是正在运行还是被切换出了内存，等等。为了获得一个有代表性的值，应该重复执行多次 BENCHMARK()查询。

- BIT_COUNT (*n*)

  用 BIGINT 值（64 位整数）返回参数 n 中被设置为 1 的二进制位的个数。

```
BIT_COUNT(0)                                    → 0
BIT_COUNT(1)                                    → 1
BIT_COUNT(2)                                    → 1
BIT_COUNT(7)                                    → 3
BIT_COUNT(-1)                                   → 64
BIT_COUNT(NULL)                                 → NULL
```

- BIT_LENGTH (*str*)

  返回给定参数 *str* 以二进制位为计算单位的长度。如果 *str* 参数是 NULL，则返回 NULL。

```
BIT_LENGTH('abc')                               → 24
BIT_LENGTH('a long string')                     → 104
BIT_LENGTH(CONVERT('abc' USING ucs2))           → 48
```

- CONNECTION_ID ()

  返回 MySQL 服务器分配给当前客户的连接标识符，每个客户都会分配到一个独一无二的连接标识符。

```
CONNECTION_ID()                                 → 10146
```

- CURRENT_USER ()

  连接 MySLQ 服务器时，MySQL 服务器会使用 mysql.user 数据表里的某个特定的账户数据行对你的连接进行身份验证。CURRENT_USER()函数将以一个'user_name@host_name'格式的 utf8 字符串的形式返回那个数据行的 User 和 Host 数据列的值。在调用这个函数的时候，函数名后面的括号可以省略。

```
CURRENT_USER()                                  → 'sampadm@localhost'
SUBSTRING_INDEX(CURRENT_USER(),'@',1)           → 'sampadm'
```

可以用 CURRENT_USER()函数去查看 MySQL 服务器认为你到底是谁，也许不是你在连接 MySQL 服务器时给出的用户名不一样，因为你有可能“碰巧”通过了服务器使用其他账户对你进行的身份验证。需要特别注意的是，如果 MySQL 认为你是一个匿名用户，这个函数的返回值里的用户名部分将是空的。USER()函数返回值里的用户名部分永远是你在建立连接时给出的用户名。

- DATABASE ()

  以 utf8 字符串的形式返回默认数据库的名字。如果没有默认数据库。则返回 NULL。

```
DATABASE()                                        → 'sampdb'
```

- FOUND_ROWS ()

  返回刚执行的那条 SELCT 语句在没有 LIMIT 子句的情况下返回的数据行行数。比如说，下面这条语句最多只能返回 10 个数据行：

```
mysql> SELECT * FROM mytbl LIMIT 10;
```

  而下面这些语句可以查出前一条 SELECT 语句在没有 LIMIT 子句的情况下会返回多少个数据行：

```
mysql> SELECT SQL_CALC_FOUND_ROWS * FROM mytbl LIMIT 10;
mysql> SELECT FOUND_ROWS();
```

- DEFAULT (*col_name*)

  虽然 INSERT 语句允许使用关键字 DEFAULT 来明确表明你想把某个数据列的默认值插入到新数据行里，但这个关键字在某些特定的表达式或其他上下文里是不允许使用的。比如说，MySQL 不允许在 UPDATE 语句里使用 DEFAULT 关键字把某个数据列重置为默认值。DEFAULT()函数可以在这类场合帮到你。给定一个数据列的名字，DEFAULT()函数将返回该数据列的默认值。

```
UPDATE counts SET counter = DEFAULT(counter)
WHERE max_time > expire_time;
```

- INET_ATON (*str*)

  把 *str* 参数给定的点格式的 IP 地址字符串转换为整数形式并返回。如果 *str* 参数不是一个合法的 IP 地址，则返回 NULL。

```
INET_ATON('64.28.67.70')                       → 1075594054
INET_ATON('255.255.255.255')                   → 4294967295
INET_ATON('256.255.255.255')                   → NULL
INET_ATON('www.mysql.com')                     → NULL
```

- INET_NTOA (*n*)

  把 *n* 参数给定的整数形式的 IP 地址转换为点格式字符串并返回。如果 *n* 参数值不能被转换为一个合法的 IP 地址，则返回 NULL。

```
INET_NTOA(1075594054)                          → '64.28.67.70'
INET_NTOA(2130706433)                          → '127.0.0.1'
```

- LAST_INSERT_ID()

  LAST_INSERT_ID(*expr*)

  如果不带任何参数，这个函数将返回在当前服务器会话期间最近自动生成的 AUTO_INCREMENT 值。如果此前尚未生成过这样的值，则返回 0。如果带有 *expr* 参数，这个函数将返回该参数

的值，并把它视为最近自动生成的 AUTO_INCREMENT 值，这就使得我们可以按照自己的预想生成 AUTO_INCREMENT 序列。

第 3 章对这个问题做了更详细的讨论。对于 LAST_INSERT_ID()函数的这两种调用形式，其返回值都将由 MySQL 服务器按照“每个连接一个”的原则记录在案，不会因为其他客户的操作而发生变化，即使其他客户所执行的操作导致 MySQL 为它们自动生成了一个新的 AUTO_INCREMENT 值。

- LOAD_FILE (*file_name*)

  读取 *file_name* 文件并把它的内容返回为一个字符串。这个文件必须存放在 MySQL 服务器里，必须以一个绝对（完整）路径名的形式给出，必须是全局可读的，以确保你不会读取受保护的文件。如果 secure_file_priv 系统变量的值不为空，该变量值应该是一个目录，而你打算读取的文件必须存放在该目录里。因为这个文件必须存放在服务器里，所以你还必须具备 FILE 权限。只要有一项条件没有满足，LOAD_FILE()函数就将返回 NULL。

- MASTER_POS_WAIT(*log_file*,*pos*[,*timeout*])

  这个函数主要用于测试复制机制中的服务器。当你在一台从服务器上执行这个函数的时候，它将阻断从服务器上的其他操作，让从服务器专注于读取和处理来自主服务器的事件，直到到达 *log_file* 和 pos 参数所给定的复制位置为止。如果还给出了可选的 *timeout* 参数，它将被解释为一段以秒为单位的倒计时时间，其作用是为 MASTER_POS_WAIT()函数所导致的阻断时间设置一个上限。如果这个参数值小于或等于 0，表示没有时间限制。

  MASTER_POS_WAIT()函数的返回值表示距离给定的复制位置还有多少个日志文件事件需要处理。如果从服务器已经到达该位置，这个函数将立刻返回 0。如果这个函数的返回值是-1，可能是因为它设置的倒计时时间已到、发生了错误、主服务器信息未被初始化，等等。如果这个函数的返回值是 NULL，则表明从服务器的 SQL 线程在该函数所设置的等待期内没有运行或是已停止。

- NAME_CONST (*name*, *value*)

  这个函数仅供 MySQL 内部使用（例如，为了把语句写入二进制日志）。它将返回 *name* 参数所给定的数据列名字和 *value* 参数，而这两个参数必须是常数。NAME_CONST()函数是从 MySQL 5.0.12 版开始引入的。

- ROW_COUNT ()

  这个函数是 MySQL C API 函数库中的 mysql_affected_rows()函数在 SQL 语言里的实现。它将返回受到前一条 SQL 语句影响的数据行的个数，也就是该语句实际插入、删除或更新的数据行的个数。如果这个函数的返回值是-1，则表明前一条语句是一条 SELECT 语句（或者是返回一个结果集的其他语句）或是该语句在执行时发生了错误。

  ROW_COUNT()函数是从 MySQL 5.0.1 版开始引入的。

- SCHEMA ()

  这个函数是 DATABASE()的同义词，它是从 MySQL 5.0.2 版开始引入的。

- SESSION_USER ()

  这个函数是 USER()的一个同义词。

- SLEEP (*seconds*)

  让当前客户休眠 *seconds* 秒并返回 0。如果在休眠时被意外打断，则返回 1。*seconds* 参数可

以有小数部分。这个函数从 MySQL 5.0.12 版开始引入的。

- SYSTEM_USER ()

  这个函数是 USER()的一个同义词。

- USER ()

  返回一个 utf8 字符串，表示当前客户在连接 MySQL 服务器时给出的用户名和该客户所在主机的名字，该字符串的格式是'*user_name@host_name*'。请特别注意，这个函数的返回值是一个 utf8 字符串；在把这个值传递给某个需要多个字符串参数的函数时，一定要注意这个问题以避免触发“排序方式不匹配”错误。

```
USER()                           → 'paul@localhost'
SUBSTRING_INDEX(USER(),'@',1)    → 'paul'
SUBSTRING_INDEX(USER(),'@',-1)   → 'localhost'
```

- UUID ()

  返回一个“全局的独一无二的标识符”（universal unique identifier）。UUID()函数每次调用的返回值应该是不相同的。严格地讲，它并不能绝对保证每一个返回值都是独一无二的，只是出现重复值的概率非常非常低而已。

```
UUID()                           → '4550868e-3c1f-1027-9cc8-78fa7f8d46b6'
UUID()                           → 'cbb9ad76-3d10-1027-8c06-349c71608da3'
```

  这个函数的返回值是用 128 位整数生成的一个由 5 组十六进制数构成的 utf8 字符串。前 4 个部分应该是唯一时间值，最后一个部分应该是唯一地点标识值。

  前 3 个部分来自一个时间戳，第四个部分用于保证这个时间戳的唯一性（例如因采用夏日制而导致的时间变化）而特意增加的。第五个部分是一个 IEEE 802 结点值，它是用一个在服务器主机上具备唯一性的值（例如网络接口地址）生成的。如果无法获得这样的独一无二的值，则使用一个随机生成的 48 位整数代替。

- VERSION ()

  返回一个 utf8 字符串来描述当前 MySQL 服务器的版本。

```
VERSION()                        → '5.1.25-rc-log'
```

  这个函数的返回值在版本号的后面往往还会有一个或多个后缀。下面是比较常见的后缀。

- -alpha、-beta或-rc。表明当前MySQL版本的稳定性。如果没有这些后缀，则表明那是一个已经足够稳定的General Availvbility（适用于各种实际应用）版本。
- -debug。意思是该MySQL服务器正以调试模式运行。
- -embedded。表明这是一个嵌入式MySQL服务器，即libmysqld。
- -log。意思是启用了日志功能。

附录 D

系统变量、状态变量和用户变量使用指南

本附录介绍以下几种 MySQL 变量：

- ❑ 提供关于服务器配置信息的系统变量；
- ❑ 每个客户都有的会话级系统变量；
- ❑ 提供关于服务器当前操作状态信息的状态变量；
- ❑ 由用户定义、赋值和在表达式里使用的用户变量。

如果没有特别注明，代表缓冲区尺寸或长度的变量的值通常以字节为单位。

除非另有说明，这里列出的变量至少从 MySQL 5.0.0 版开始就已经出现了。在 MySQL 5.0.0 版以后新增加的或是变化了的变量会另有说明。

D.1 系统变量

系统变量提供关于服务器的配置和能力的信息。绝大多数系统变量都可以在服务器启动时设置，有许多还可以在服务器运行时动态地设置。在介绍每个变量时，我们会在变量名后面的括号里给出这方面的信息。

- ❑ 对于那些可以在服务器启动时设置的变量，你将在括号里看到“启动”后面跟着“直接设置”或一个选项。“直接设置”就是可以直接在命令行上或者是在选项文件里使用与变量名同名的选项来设置该变量。（F.1.2节中的第2小节描述了这么做的语法。）否则，将需要使用“启动”后面列出的选项来设置。比如说，`storage_engine`变量需要使用`--default-storage-engine`选项来设置。如果“启动”后面跟着一个选项，它的含义可以在附录F对`mysqld`程序的描述里查到。
- ❑ 对于那些可以在服务器运行时动态设置的变量，你将在括号里看到“运行时”后面跟着“全局”和/或“会话”，表示该变量有一种`GLOBAL`形式和/或一种`SESSION`形式。

有些系统变量只存在于会话级。D.2 节描述了这些变量。

系统变量可以用 `SHOW VARIABLES` 语句或通过执行 `mysqladmin variables` 命令的办法来查看。各全局级变量的值还可以用 `SELECT @@GLOBAL.`*`var_name`* 语句来查看，而会话级变量的值还可以用 `SELECT @@SESSION.`*`var_name`* 或 `SELECT @@`*`var_name`* 语句来查看。从 MySQL 5.1.12 版开始，还可

以通过查看 INFORMATION_SCHEMA 数据库里的 GLOBAL_VARIABLES 和 SESSION_VARIABLES 数据表的办法来获取关于系统变量的信息。

关于如何在服务器运行时动态设置或查看系统变量的信息，参见 12.6.1 节。

系统变量的名字不区分大小写。

这里描述的变量有些只在特定的配置情况下才会出现。比如说，以 innodb_开头的许多变量都只在可以选用 InnoDB 存储引擎时才会存在。有几个与 Falcon 存储引擎有关的变量在本附录里没有提到，因为它们目前仍在开发中，还不够稳定。

- auto_increment_increment（启动：直接设置；运行时：全局、会话）
  MySQL 服务器为 AUTO_INCREMENT 数据列每次生成一个新序列值时的递增量。默认值是 1，取值范围的 1 到 65535。这个变量是从 MySQL 5.0.2 版开始引入的。
- auto_increment_offset（启动：直接设置；运行时：全局、会话）
  AUTO_INCREMENT 序列编号的初始值。默认值是 1，取值范围的 1 到 65535。这个变量是从 MySQL 5.0.2 版开始引入的。
- auto_sp_privileges（启动：直接设置；运行时：全局）
  当这个变量是 1（默认值）时，MySQL 服务器将在你创建一个存储例程时自动授予你 EXECUTE 和 ALTER ROUTINE 权限，让你以后可以执行、修改或删除该例程。当你删除该例程时，MySQL 服务器将自动收回那些权限。如果 automatic_sp_privileges 变量的值是 0，上述自动授予/收回权限的机制将不起作用。这个变量是从 MySQL 5.0.3 版开始引入的。
- back_log（启动：直接设置）
  当 MySQL 服务器正在处理当前连接时，后来的连接请求将排队等候处理。这个变量用来设置那个队列所能容纳的连接请求的最大个数。
- basedir（启动：直接设置）
  MySQL 软件的安装根目录的路径名。
- binlog_cache_size（启动：直接设置；运行时：全局）
  二进制日志的缓存区长度。在被转储清除到二进制日志之前，构成某个事务的 SQL 语句都缓存在这个缓存区里。（只有事务被提交或是包括对非事务型数据表进行更新的语句，把语句写入二进制日志的操作才会发生。如果某个事务只对事务型数据表进行更新并被回滚了，构成该事务的语句将被丢弃。）
- binlog _format（启动：直接设置；运行时：全局）
  二进制日志的记录格式。其可取值包括 STATEMENT、ROW 或（从 MySQL 5.1.8 版开始）MIXED，它们分别代表着基于语句的日志格式、基于数据行日志格式和混合型日志格式。如果你选择使用 MIXED 格式，MySQL 服务器将根据具体情况自动切换使用基于语句的和基于数据行的日志格式。从 MySQL 5.1.12 版开始，这个变量的默认值是 MIXED。这个变量是从 MySQL 5.1.5 版开始引入的，但从 5.1.8 版才开始允许在运行时动态设置。
- bulk_insert_buffer_size（启动：直接设置；运行时：全局、会话）
  用来优化 MyISAM 数据表的批量插入语句的缓存区的尺寸。这包括 LOAD DATA 语句、一次插入多个数据行的 INSERT 语句和 INSERT INTO ... SELECT 语句。把这个变量设置为零将禁用这种优化。
- character_set_client（运行时：全局、会话）

客户向服务器发送 SQL 语句时使用的字符集。

- character_set_connection（运行时：全局、会话）
  客户–服务器连接的字符集。这个字符集用来解释字符串常数里的各个字符（带有特殊前导符的字符不包括在内）和经“从数值到字符串”转换而得到的结果。
- character_set_database
  默认数据库（如果有的话）的字符集。如果没有默认的数据库（比如说，如果客户在连接MySQL服务器时没有指定默认数据库），这个变量将被设置为character_set_server系统变量的值。每当用户选择了一个不同的默认数据库，MySQL服务器就会自动设置character_set_database。
- character_set_filesystem（启动：直接设置；运行时：全局、会话）
  文件系统的字符集。这个字符集用来解释 SQL 语句里的文件名字符串，如 LOAD DATA 语句里的数据文件的名字。服务器在访问文件之前会先把文件名从 character_set_client 变量指定的字符集转换为 character_set_filesystem 变量指定的字符集。默认值是 binary（不转换）。这个变量是从 MySQL 5.0.19/5.1.6 版开始引入的。
- character_set_results（运行时：全局、会话）
  由服务器发送到客户的结果集的字符集。
- character_set_server（启动：直接设置；运行时：全局、会话）
  服务器的默认字符集。
- character_set_system
  系统字符集，它的值总是utf8。这个字符集用来解释各种元数据，如数据库、数据表和数据列的名字。DATABASE()、CURRENT_USER()、USER()和VERSION()等函数也要用到这个字符集。
- character_set_dir（启动：直接设置）
  用来存放字符集文件的子目录。
- collation_connection（运行时：全局、会话）
  连接字符集的排序方式。
- collation_database
  数据库字符集（如果有的话）的排序方式。如果没有默认的数据库（比如说，如果客户在连接MySQL服务器时没有指定默认数据库），这个变量将被设置为collation_server系统变量的值。每当用户选择了一个不同的默认数据库，MySQL 服务器就会自动设置collation_database变量。
- collation_server（启动：直接设置；运行时：全局、会话）
  服务器字符集的排序方式。
- completion_type（启动：直接设置；运行时：全局、会话）
  事务的完成类型。如果这个变量的值是 0（默认值），COMMIT 和 ROLLBACK 语句不受任何影响。如果这个变量的值是 1，将导致 COMMIT 和 ROLLBACK 语句分别等价于 COMMIT AND CHAIN 和 ROLLBACK AND CHAIN 语句。如果这个变量的值是将导致 COMMIT 和 ROLLBACK 语句分别等价于 COMMIT AND RELEASE 和 ROLLBACK AND RELEASE 语句。AND CHAIN 的含义是在一个事务结束时，服务器将以同样的隔离级别自动开始一个新的事务；AND RELEASE 的含义是在一个事务结束时，服务器将结束当前连接。这个变量是从 MySQL 5.0.3 版开始引入的。

- concurrent_insert（启动：直接设置；运行时：全局）
  是否允许并发插入操作，即 MySQL 服务器是否允许在有 SELECT 语句正在访问一个 MyISAM 数据表（没有空洞）的同时对该数据表执行 INSERT 语句。如果这个变量的值是 0，禁用并发插入操作。如果这个变量的值是 1，启用并发插入操作。如果这个变量的值是 2（只能在 MySQL 5.0.6 及更高版本里使用），则对所有的 MyISAM 数据表启用并发插入操作，不管它们的数据文件里是否有空洞；此时，如果数据表正在被使用，新数据行将被添加到数据表的末尾。这个变量的默认值是 1，但可以在启动 MySQL 服务器时通过直接设置这个变量或是使用 --skip-concurrent_insert 选项的办法禁用。
- connect_timeout（启动：直接设置；运行时：全局）
  MySQL 服务器程序 mysqld 在开始建立连接时等待客户发来数据包的时间，以秒为单位。从 MySQL 5.0.52/5.1.23 版开始，这个变量的默认值是 10 秒；在更早的版本里，它的默认值是 5 秒。
- datadir（启动：直接设置）
  MySQL 数据目录的路径名。
- date_format
  这个变量目前尚未投入使用。
- datetimne_format
  这个变量目前尚未投入使用。
- default_week_format（启动：直接设置；运行时：全局、会话）

这个变量用来设置 WEEK()或 YEARWEEK()函数在不带可选的 mode 参数时使用的默认模式值。

- delay_key_write（启动：直接设置；运行时：全局）
  MySQL 服务器在遇到使用 DELAY_KEY_WRITE 选项创建出来的 MyISAM 数据表时是否真的进行延迟键写操作。这个变量有 3 种可取值。
  - ON（默认值）。让服务器根据DELAY_KEY_WRITE选项的值进行操作。如果数据表是用DELAY_KEY_WRITE=1选项创建的，键的写操作将被延迟；如果数据表是用DELAY_KEY_WRITE=0选项创建的，不延迟。
  - OFF。对任何数据表的键写操作都不延迟，不管它们当初是如何定义的。
  - ALL。对任何数据表的键写操作都进行延迟，不管它们当初是如何定义的。
- delayed_insert_limit（启动：直接设置；运行时：全局）
  在处理 INSERT DELAYED 语句的时候，MySQL 服务器每插入 delayed_insert_limit 个数据行，就会去看看是否有新到的 SELECT 语句正排队等待着对有关的数据表进行检索。如果有，则挂起插入操作让检索操作优先执行。
- delayed_insert_timeout（启动：直接设置；运行时：全局）
  在处理 INSERT DELAYED 语句的时候，当把队列里的数据行全都插入到有关的数据表里之后，MySQL 会等待 delayed_insert_timeout 秒，看有没有新的 INSERT DELAYED 数据行到达。如果有，则继续插入；如果没有，则结束这次插入操作。
- delayed_queue_size（启动：直接设置；运行时：全局）
  在被实际插入到各有关数据表里去之前，INSERT DELAYED 数据行将在一个队列里等待 MySQL 来处理它们，delayed_query_size 就是这个队列所能容纳的数据行的最大个数。当这个队列满时，后续的 INSERT DELAYED 语句将被阻塞直到这个队列里有容纳它们的空间为止。

- div_precision_increment（启动：直接设置；运行时：全局、会话）
  在使用“/”操作符对两个精确数值做除法运算的时候，这个变量决定着需要增加几位数字来满足计算精度要求。比如说，当这个变量的值是 4 或 6 时，0.1/0.7 的结果将分别是 0.14286 和 0.1428571。这个变量的取值范围是 0 到 30，默认值是 4。这个变量是从 MySQL 5.0.6 版开始引入的。
- event_scheduler（启动：直接设置）
  事件调度器的状态。这个变量的可取值是 OFF、ON 或 DISABLED。如果在启动 MySQL 服务器时把这个变量设置为 DISABLED，事件调度器的状态将无法在 MySQL 服务器正在运行时改变。如果在启动 MySQL 服务器时把这个变量设置为 ON 或 OFF，在 MySQL 服务器正在运行时就只能让事件调度器的状态在这两个值之间改变。这个变量是从 MySQL 5.1.6 版开始引入的。
- expire_logs_days（启动：直接设置；运行时：全局）
  这个变量的默认值是 0。如果把它设置为一个非零值，MySQL 服务器将自动删除在 expire_logs_days 天之前创建的二进制日志文件，并对二进制日志索引文件进行相应的更新。MySQL 服务器将在它每次启动和它每次打开一个新的二进制日志文件时进行这种失效检查。
- flush（启动：使用--flush；运行时：全局）
  如果这个变量的值是 ON，MySQL 服务器将在每个修改操作完成后立刻转储清除数据表；如果这个变量的值是 OFF，则不这样做。默认值是 OFF。在命令行上使用--flush 选项将启用“修改后立刻转储清除”功能。
- flush_time（启动：直接设置；运行时：全局）
  如果这个变量是一个非零值，那么每隔 flush_time 秒，MySQL 就会关闭数据表以便把尚未写入磁盘的修改写入磁盘。如果你的系统不够稳定，经常死机或重启，用这个办法来强行更新数据表，可以减少数据表受损或数据丢失的概率，但代价是性能降低。这个变量在 Unix 系统上的默认值是 0，在 Windows 系统上的默认值是 1800 秒（30 分钟）。
- ft_boolean_syntax（启动：直接设置；运行时：全局）
  在 IN BOOLEAN MODE 模式下进行 FULLTEXT 搜索所允许使用的操作符的清单。
- ft_max_word_len（启动：直接设置）
  允许包括在 FULLTEXT 索引里的单词的最大长度，长于这个长度的单词将被忽略。如果改变了这个变量的值，就必须重建所有数据表里的 FULLTEXT 索引。这个变量的默认值是 84。
- ft_min_word_len（启动：直接设置）
  允许包括在 FULLTEXT 索引里的单词的最小长度，短于这个长度的单词将被忽略。如果改变了这个变量的值，就必须重建所有数据表里的 FULLTEXT 索引。这个变量的默认值是 4。
- ft_query_expansion_limit（启动：直接设置）
  这个变量用于使用 WITH QUERY EXPANSION 子句进行的全文检索操作。每次搜索分两个阶段进行，第一阶段找到的相关性最高的前 ft_query_expansion_limit 个匹配将被用来进行第二阶段的搜索。
- ft_stopword_file（启动：直接设置）
  FULLTEXT 索引的“休止词”（stopword）文件。这个变量的默认值使用的是 MySQL 服务器内建的休止词清单。把这个变量设置为空字符串将禁用休止词功能。如果改变了这个变量的值或休止词文件的内容，就必须重建所有数据表里的 FULLTEXT 索引。

- general_log（启动：直接设置；运行时：全局）
  是否启用“普通查询”（general query）日志功能。（如果启用了这项功能，日志信息将被输出到 log_output 变量所指定的目的地。）这个变量是从 MySQL 5.1.12 版开始作为 log 变量的一个同义词被引入的。
- general_log_file（运行时：全局）
  “普通查询”日志文件的名字。这个变量只在你指定了日志信息的输出目的地时才起作用。这个变量是从 MySQL 5.1.12 版开始引入的。
- group_concat_max_len（启动：直接设置；运行时：全局、会话）
  GROUP_CONCAT()函数的返回值的长度上限（默认值是 1024）。
- have_compress
  MySQL服务器需要有zlib压缩库的支持才能实现COMPRESS()和UNCOMPRESS()函数。这个变量用来表明zlib库是否可用；如果它不可用，用户就无法使用那两个函数。
- have_crypt
  MySQL服务器需要有crypt()系统调用的支持才能实现CRYPT()函数。这个变量用来表明crypt()系统调用是否可用；如果它不可用，用户就无法使用CRYPT()函数。
- have_dynamic_loading
  MySQL服务器是否支持动态地加载各种插件。这个变量是从MySQL 5.1.10版开始引入的。
- have_engine_name
  每个have_engine_name变量（例如have_innodb）表明MySQL服务器是否支持某种特定的存储引擎。并非每一种存储引擎都有一个这样的变量。对于有这样一个变量的存储引擎，YES值的含义是“有这种存储引擎，可以使用”，NO值的含义是“没有这种存储引擎”。在MySQL 5.1.18版之前，还有一个DISABLED值用来表明“MySQL服务器支持这种存储引擎，但它在服务器启动时已被禁用”。
- have_geometry
  如果MySQL服务器允许用户使用空间数据类型，这个变量的值将是YES；否则，这个变量的值将是NO。
- have_openssl
  如果MySQL服务器允许客户使用SSL来建立加密连接，这个变量的值将是YES；否则，这个变量的值将是NO。从MySQL 5.0.38/5.1.17版开始，have_ssl和have_openssl变量互为同义词。
- have_query_cache
  如果可以使用查询缓存区，这个变量的值将是YES；否则，这个变量的值将是NO。
- have_raid
  这个变量的值总是NO。对RAID数据表的支持是一种很“古老”的功能，已经从MySQL 5.0版开始被彻底去除了。
- have_rtree_keys
  如果SPATIAL索引可以被创建为RTREE索引，这个变量的值将是YES；否则，这个变量的值将是NO。
- have_ssl

如果MySQL服务器允许客户使用SSL来建立加密连接，这个变量的值将是YES；否则，这个变量的值将是NO。这个变量是从MySQL 5.0.38/5.1.17版开始作为have_openssl变量的一个同义词被引入的。

- have_symlink

  这个变量的可取值是YES或NO，但它们的含义要取决于具体的系统平台。在Unix系统上，这个变量用来表明是否可以为MyISAM数据表创建符号链接。在Windows系统上，它用来表明是否可以为数据库创建符号链接。

- hostname

  MySQL服务器的主机名。MySQL服务器会在启动时自动确定这个变量的值。这个变量是从MySQL 5.0.38/5.1.17版开始引入的。

- init_connect（启动：直接设置；运行时：全局）

  如果给出，这个变量的值应该是一个非空值，这个非空值应该由一条或多条以分号分隔的 SQL 语句构成，那些语句将在每个客户连接到 MySQL 服务器时自动执行。这个变量可以用来为连接到 MySQL 服务器的客户建立一个初始会话环境。不过，对于具备 SUPER 权限的用户，init_connect 变量将被忽略，这是为了避免这个变量的值里包含有一条不正确或是不恰当的语句而导致管理员用户无法连接到 MySQL 服务器去改正错误。

- init_file（启动：直接设置）

  如果给出，这个变量的值应该是一个非空值，这个非空值应该是一个文件名，其内容是 MySQL 服务器在启动时将自动执行的 SQL 语句。在这个文件里，每条语句占用一行。

- init_slave（启动：直接设置；运行时：全局）

  如果给出，这个变量的值应该是一个非空值，这个非空值应该由一条或多条以分号分隔的 SQL 语句构成，那些语句将在复制机制中的从服务器每次启动其 SQL 线程时被自动执行。

- innodb_adaptive_hash_index（启动：直接设置）

  启用或者禁用 InnoDB 存储引擎的自适应散列索引。这个变量的默认值将启用该索引，但可以通过使用--skip-innodb_adaptive_hash_index 选项启动 MySQL 服务器的办法加以禁用。这个变量是从 MySQL 5.0.52/5.1.24 版开始引入的。

- innodb_additional_mem_pool_size（启动：直接设置）

  InnoDB 存储引擎用来存放各种内部数据结构的内存池的长度。

- innodb_autoextend_increment（启动：直接设置；运行时：全局）

  当某个自扩展表空间快要被填满时，InnoDB 存储引擎将自动使用这个变量的值作为递增量（以 MB 为单位）去加大那个表空间的尺寸。这个变量的默认值是 8，最大值是 1000。

- innodb_buffer_pool_awe_mem_mb（启动：直接设置）

  这变量只与支持 Address Windows Extensions 的 32 位 Windows 系统有关，它的值是分配给 InnoDB 缓冲池使用的 AWE 内存的长度（以 MB 为单位）。这个变量的最大可取值是 63000。如果设置了这个变量，MySQL 服务器将按照 innodb_buffer_pool_size 变量给出的长度在 mysqld 程序的地址空间里划出一个“窗口”供 InnoDB 存储引擎去访问那些被缓存在 AWE 内存里的信息。

- innodb_buffer_pool_size（启动：直接设置）

  InnoDB 缓存区的长度，其中存放着 InnoDB 数据表的数据和索引。

- innodb_checksums（启动：直接设置）
  如果 InnoDB 数据表的校验和计算功能已被启用，这个变量的值将是 ON；否则，这个变量的值将是 OFF。这个变量是从 MySQL 5.0.3 版开始引入的。
- innodb_commit_concurrency（启动：直接设置；运行时：全局）
  这个变量决定着多少个线程可以同时提交。默认值是 0，其含义是“没有限制”。这个变量是从 MySQL 5.0.12 版开始引入的。
- innodb_concurrecy_tickets（启动：直接设置；运行时：全局）
  当一个线程想进入 InnoDB 时，只有在线程的数量小于 innodb_commit_concurrency 变量所设置的上限时它才能成功。否则，该线程将排队等候直到线程的数量降低到那个上限以下。一旦线程被允许进入，它将可以不受限制地离开和重新进入 InnoDB，这种自由往返的最大次数由 innodb_concurrecy_tickets 变量的值决定。这个变量是以 MySQL 5.0.3 开始引入的。
- innodb_data_file_path（启动：直接设置）
  InnoDB 表空间组件文件的定义。
- innodb_data_home_dir（启动：直接设置）
  子目录路径名，相对于存放 InnoDB 表空间组件文件的位置。如果这个变量是空值，组件文件名将被解释为绝对路径名。
- innodb_doublewrite（启动：直接设置）
  表明 InnoDB 双写缓冲区是否已被启用，可取值是 ON 和 OFF。默认值是 ON。这个变量是从 MySQL 5.0.3 版开始引入的。
- innodb_fast_shutdown（启动：直接设置；运行时：全局）
  InnoDB 是否将使用它的快速关机方法，该方法省略了它在正常关机时会进行的几个操作步骤。可取值是 0 和 1。
- innodb_file_io_threads（启动：直接设置）
  InnoDB 用来完成文件 I/O 操作的线程的个数。这个变量目前只在 Windows 系统上有实际效果。在某些场合里，从默认值 4 开始适当加大这个变量的值可以改善性能。
- innodb_file_per_table（启动：直接设置）
  如果这个变量被设置为 0（默认值），InnoDB 将在其共享表空间里创建新数据表。如果这个变量被设置为 1，InnoDB 将为每个新数据表分别创建一个专用表空间：在数据库目录里为每一个新数据表单独创建一个.ibd 文件来存放该数据表的内容。这个变量只影响新数据表是如何创建的，不影响 InnoDB 存储引擎对现有数据表的访问。不管这个变量被设置成什么，InnoDB 存储引擎总是可以访问共享表空间或专用表空间里的现有数据表。
- innodb_flush_log_at_trx_commit（启动：直接设置；运行时：全局）
  这个选项控制着 InnoDB 存储引擎的日志转储清除行为，它的可取值如表 D-1 所示。

表　D-1

取　值	含　义
0	每隔一秒写一次日志，同时转储到相应的磁盘文件
1	每次提交事务时写一次日志，同时转储到相应的磁盘文件
2	每次提交事务时写一次日志，但每隔一秒刷新一次相应的磁盘文件

请注意，如果没有把这个变量设置为 1，InnoDB 将不能保证 ACID 特性。在最坏的情况下，在发生崩溃的前一秒内进行的事务都可能丢失。

- innodb_flush_method（启动：直接设置）

  这个变量给出 InnoDB 用来转储文件的方法。它只适用于 Unix 系统。可取值包括：fdatasync（使用 sync()来转储数据文件和日志文件）、O_DSYNC（使用 sync()转储数据文件，使用 O_SYNC 打开和转储日志文件）和 O_DIRECT（使用 sync()转储数据文件和日志文件，视情况选用 O_DIRECT 或 directio()打开数据文件）。默认值是 fdatasync。在 Windows 系统上。这个变量的值永远是 async_unbuffered。

- innodb_force_recovery（启动：直接设置）

  这个变量的值通常是 0，但可以被设置为从 1 到 6 的某个值，以使 MySQL 服务器即使在 Innodb 恢复失败的情况下也能在崩溃后再次启动。关于如何使用这个变量的详细描述见 14.7.4 节。

- innodb_lock_wait_timeout（启动：直接设置）

  在准备执行一次事务时，如果 InnoDB 在等待了 innodb_lock_wait_timeout 秒后还没有获得所申请的数据锁，InnoDB 就将回滚这次事务。

- innodb_locks_unsafe_for_binlog（启动：直接设置）

  如果 InnoDB 存储引擎在进行索引搜索和扫描时的"下一个键锁定"（next-key locking）功能已被禁用，这变量的值将是 ON；否则，这个变量的值将是 OFF。默认值是 OFF，即启用"下一个键锁定"功能。在普通情况下，当 InnoDB 存储引擎锁定一个数据行的时候，它不仅会同时锁定该数据行的索引记录，还会阻止其他客户把一条新的索引记录紧挨着插入到被锁定的索引记录的前面。这被称为"下一个键锁定"，其用途是防止数据表里出现不该出现的"影子"数据行。启用 innodb_locks_unsafe_for_binlog 变量将禁用"下一个键锁定"功能，其效果是在锁定某个数据行的同时只锁定该数据行的索引记录，但不阻止其他客户把一条新的索引记录紧挨着插入到被锁定的索引记录的前面。这将导致以下后果。

  - 有些原本应该被阻塞的插入操作现在可以进行了。
  - 数据表里可能会出现不该出现的"影子"数据行。
  - InnoDB 存储引擎最多只能保证达到 READ COMMITTED 级别的隔离效果，不能保证达到可串行级别的隔离效果。
  - 从 MySQL 5.0.2 版开始，如果启用了 innodb_locks_unsafe_for_binlog 变量，InnoDB 存储引擎在开始执行一条语句时将先锁定它将要扫描的数据行（像往常一样），然后为 DELETE 或 UPDATE 语句只保留真正需要修改的数据行上的锁定。其他数据行上的锁定将在 InnoDB 存储引擎确定自己可以跳过它们之后被释放。这可以减少发生死锁现象的概率。

  innodb_locks_unsafe_for_binlog 变量只影响索引搜索和扫描操作，不影响它对外键约束条件和重复键的检查操作。

- innodb_log_arch_dir（启动：直接设置）

  这个变量目前未被使用。MySQL 5.1.21 版删除了这个变量。

- innodb_log_archive（启动：直接设置）

  这个变量目前未被使用。MySQL 5.1.18 版删除了这个变量。

- innodb_log_buffer_size（启动：直接设置）

  InnoDB 事务日志缓冲区的尺寸。默认值是 1MB。实际取值通常在 1MB 到 8MB 之间。

- innodb_log_file_size（启动：直接设置）
  每个 InnoDB 日志文件的长度。innodb_log_file_size 和 innodb_log_files_in_group 的乘积就是 InnoDB 日志的总长度。
- innodb_log_files_in_group（启动：直接设置）
  在 InnoDB 管理下的日志文件的个数。innodb_log_files_in_group 和 innodb_log_file_size 的乘积就是 InnoDB 日志的总长度。
- innodb_log_group_home_dir（启动：直接设置）
  InnoDB 日志目录的路径名，InnoDB 日志文件将被写到这个子目录里去。
- innodb_max_dirty_page_pct（启动：直接设置；运行时：全局）
  每当 InnoDB 缓冲池里的“脏”页面超过了由这个变量设定的百分比，InnoDB 存储引擎就会把被缓存的日志信息写入相应的磁盘文件。这个值应该在 0 到 100 之间。默认值是 90。
- innodb_max_purge_lag（启动：直接设置；运行时：全局）
  InnoDB 存储引擎在执行 DELETE 或 UPDATE 语句时采用的策略是先把有关的数据行标记为“待删除”，再用一个线程专门来实际清除那些“待删除”数据行。如果小批量数据行的插入和删除操作的频率差不多相同，清除线程就可能落后，进而导致许多“待删除”数据行不能及时清除而仍占用着本应该释放的空间。innodb_max_purge_lag 变量控制着如何延迟 INSERT、DELETE 和 UPDATE 语句，让它们适当放慢速度以便清除线程能够追上它们的进度。默认值是 0（不延迟）。如果把这个变量设置为一个非零值，延迟大约是((n / innodb_max_purge_lag) × 10) -5 毫秒，其中的 n 是在执行过程中会把一些数据行标记为“待删除”的事务的数量。
- innodb_mirrored_log_groups（启动：直接设置）
  InnoDB 日志文件组的个数，这个变量的值应该永远是 1。
- innodb_open_files（启动：直接设置）
  如果 innodb_file_per_table 变量被设置为 1，每一个新创建的 InnoDB 数据表都将有一个它自己专用的表空间。innodb_open_files 变量控制着需要为 InnoDB 存储引擎保留多少个文件描述符以便它能同时打开多个.ibd 文件。这个变量的最小值是 10，默认值是 300。在 innodb_open_files 变量控制下分配的文件描述符与在 open_files_limit 变量控制下分配的文件描述符互不相干：前者用来打开.ibd 文件，后者供数据表缓冲区使用。
- innodb_rollback_on_timeout（启动：直接设置）
  这个变量控制着 InnoDB 存储引擎在某个事务超出时限时的行为。如果这个变量的值是 OFF(默认值)，InnoDB 存储引擎将只回滚最后一条语句；如果这个变量的值是 ON，InnODB 存储引擎将回滚整个事务。这个变量是从 MySQL 5.0.32/5.1.15 版开始引入的。在更早的版本里，InnoDB 存储引擎将回滚整个事务。
- innodb_support_xa（启动：直接设置；运行时：全局、会话）
  如果 InnoDB 存储引擎支持 XA 事务中的两阶段提交，这个变量的值将是 ON；否则，这个变量的值将是 OFF。默认值是 ON。如果没有使用 XA 事务，把它设置为 OFF 可以改善性能。这个变量是从 MySQL 5.0.3 版开始引入的。
- innodb_sync_spin_loops（启动：直接设置；运行时：全局）
  这个变量给出了一个循环等待次数。如果线程在等待 InnoDB 存储引擎释放互斥信号时超过这个等待次数，它就会被挂起。这个变量是从 MySQL 5.0.3 版开始引入的。

- innodb_table_locks（启动：直接设置；运行时：全局、会话）
  在自动提交模式被禁用的情况下，当有 LOCK TABLE 语句试图为某个 InnoDB 数据表申请一个写操作锁时，InnoDB 存储引擎将根据这个变量的值采取行动。如果这个变量的值是 ON（默认值），InnoDB 将申请到一个内部数据表锁。如果这个变量的值是 OFF，InnoDB 将一直等到没有任何其他线程锁定那个数据表时才执行 LOCK TABLE 语句。禁用这个变量（把它设置为 OFF）可以在一定程度上防止应用程序在自动提交模式已被禁用的情况下发出 LOCK TABLE 语句时遭遇死锁。
- innodb_thread_concurrency（启动：直接设置；运行时：全局）
  为 InnoDB 存储引擎能够同时管理的线程个数设置一个上限。这个变量从 MySQL 5.0.3 版开始才成为一个可以在运行时设置的全局级系统变量。
- innodb_thread_sleep_delay（启动：直接设置；运行时：全局）
  一个以毫秒为单位的时间值，如果某个 InnoDB 线程休眠了这么长的时间，它就会被放入 InnoDB 等待队列。默认值是 10 000（10s），0 值的含义是“不休眠”。这个变量是从 MySQL 5.0.3 版开始引入的。
- interactive_timeout（启动：直接设置；运行时：全局、会话）
  如果某个交互式的客户连接在 interactive_timeout 秒内没有操作动作，MySQL 服务器就将认为该客户连接不再有保留的必要并自动关闭这个连接。对于非交互式的客户连接，MySQL 服务器将使用 wait_timeout 变量的值作为这种超时等待的时间。
- join_buffer_size（启动：直接设置；运行时：全局、会话）
  没有使用索引的和需要进行全表扫描的联结操作所使用的缓冲区的长度。
- keep_files_on_create（启动：直接设置；运行时：全局、会话）
  如果你在用来创建 MyISAM 数据表的 CREATE TABLE 语句里明确地给出了 DATA DIRECTORY 或 INDEX DIRECTORY 选项，但 MySQL 服务器发现在给定的子目录里已经存在着一个同名的数据文件或索引文件，它将返回一个错误。如果你在创建 MyISAM 数据表的时候没有使用 DATA DIRECTORY 或 INDEX DIRECTORY 选项为数据文件或索引文件指定一个存放位置，MySQL 服务器将根据 keep_files_on_create 变量的值采取行动：如果这个变量的值是 OFF（默认值），即使 MySQL 服务器发现已经存在着一个同名的.MYD 数据文件或.MYI 索引文件，它也会覆盖它们；如果这个变量的值是 ON，MySQL 服务器在发现已经存在着同名文件时将返回一个错误。这个变量是从 MySQL 5.0.48/5.1.21 版开始引入的。
- key_buffer_size（启动：直接设置；运行时：全局）
  用来缓存 MyISAM 数据表的索引块的缓冲区的长度。这个缓冲区是负责处理数据库连接的各个线程共享的。
  这个变量和另外几个与键缓存区有关的变量（key_cache_age_threshold、key_cache_block_size 和 key_cache_limit）可以被组织为一个结构化的系统变量，我们可以把它们当做那个结构变量的成员来访问。我们可以通过创建多个键缓存的办法对键字缓存区的分配和使用情况进行更细致的控制；关于这方面的详细讨论见 12.7.2 节。
- key_cache_age_threshold（启动：直接设置；运行时：全局）
  如果键缓存的热链里的某个缓存块在这个变量所设定的时间里没有被使用过，就将被移入温链。这个变量的值越大，缓存块在热链里待的时间就越长。这个变量的默认值是 200，最小值是 100。

- key_cache_block_size（运行时：全局）
  键缓存里的缓存块的长度。在默认的情况下，缓存块的单位长度是 1 024 个字节。
- key_cache_limit（运行时：全局）
  如果这个变量被设置为它的默认值 100，键缓存将使用“最近最少使用”策略来处理各缓存块的重复使用问题。如果这个变量的值小于 100，键缓存区将使用“中间插入”策略来处理，这个变量的值将被视为一个百分比数值，使用频率小于这个百分比的缓存块将被移入温链。这个变量的取值范围是 1 到 100。
- language（启动：直接设置）
  用来显示出错消息的语言。这个变量的值可以是某种语言的名字，也可以是用来保存相关语言文件的子目录的路径名。
- large_files_support
  MySQL服务器是否支持对大文件进行处理。
- large_page_size
  如果启用了大页面支持，这个变量的值将是大内存页面的长度；否则，这个变量的值是0。这个变量是从MySQL 5.0.3版开始引入的。
- large_pages（启动：使用-- large-pages选项）
  是否启用对大内存页面的支持功能。目前只有基于 Linux 操作系统的 MySQL 发行版本支持使用大页面。这个变量是从 MySQL 5.0.3 版开始引入的。
- lc_time_names（启动：直接设置；运行时：全局、会话）
  这个变量控制着 DATE_FORMAT()、DAYTIME()和 MONTHNAME()等函数将使用何种语言来显示日期和月份的名字。它的默认值是 en_US，但可以被设置为其他 POSIX 风格的语言名称，如 es_AR（西班牙语/阿根廷）或 zh_HK（汉语/中国香港）。这个变量是从 MySQL 5.0.25/5.1.12 版开始引入的。
- license
  MySQL服务器所使用的许可证的类型。比如说，对于采用GPL许可证发行的MySQL服务器，这个变量的值将是GPL。
- local_infile（启动：直接设置；运行时：全局）
  在 LOAD DATA 语句里是否允许使用 LOCAL 关键字。
- locked_in_memory（启动：使用--memlock选项）
  MySQL 服务器在启动后是否一直驻留在内存里。
- log（运行时：全局）
  查询日志功能是否已经启用。从 MySQL 5.1.12 版开始，这个变量和 general_log 变量是同义词，但 log 变量在 MySQL 5.1.23 版之前不能在 MySQL 服务器正在运行时设置。
- log_bin
  二进制日志功能是否已经启用。
- log_bin_trust_function_creators（启动：直接设置；运行时：全局）
  一般来说，存储函数的创建和修改需要你具备 CREATE ROUTINE 和 ALTER ROUTINE 权限。但如果启用了二进制日志功能且 log_bin_trust_function_creators 变量被设置为 0（默认值），你还必须具备 SUPER 权限，而且你打算修改的存储函数还必须是确定的（意思是“不修

改任何数据”)。如果你想取消上述额外要求，请把这个变量设置为 1。这个变量是从 MySQL 5.0.16 版开始引入的。(在 5.0.6 到 5.0.15 版的 MySQL 软件里，这个变量的名字是 log_bin_trust_ routine_creators 并且对存储过程也有影响。)

- log_error（启动：直接设置）
  错误日志文件的名字。如果这个值为空，MySQL 服务器将把出错消息写到控制台终端。
- log_output（启动：使用--log-output选项；运行时：全局）
  “普通查询”日志和“慢查询”日志的输出目的地——如果这些日志已经启用的话。这个变量的值是一组以逗号为分隔符的输出目的地。允许使用的目的地是 TABLE、FILE 和 NONE。如果给出这个值，NONE 将禁用日志功能并优先于任何其他的值。这个变量是从 MySQL 5.1.6 版开始引入的。
  我们可以在运行时动态设置 general_log 或 slow_log 系统变量的办法来启用或是禁用相应的日志功能，还可以通过动态设置 general_log_file 或 slow_guery_log_file 系统变量的办法来改变相应的日志文件的名字。
- log_queries_not_using_indexs（启动：使用--log-queries-not-using-indexs选项；运行时：全局）
  是否要把没有使用索引的查询命令记载到“慢查询”日志里。这个变量是从 MySQL 5.0.23/5.1.11 版开始引入的。
- log_salve_updates（启动：直接设置）
  在接收到来自其主服务器的二进制日志的数据更新操作时，复制（replication）机制中的从服务器将根据这个变量来决定是否要把那些修改记载到它自己的二进制日志里。在默认的情况下，从服务器上的“修改”日志功能是被禁用的。但在建立一个复制链的时候，你将需要把前一个从服务器配置为后一个从服务器的主服务器，而这就需要在前一个从服务器上启用其“修改”日志功能。
- log_slow_queries
  是否启用“慢查询”日志功能。从MySQL 5.1.23版开始，这个变量可以在运行时动态地设置以启用或禁用“慢查询”日志功能。
- log_warnings（启动：直接设置；运行时：全局、会话）
  记录级别。把不属于“致命错误”类别的警告/出错消息记载到“错误”日志里的记录级别。如果这个变量的值是 0，不记载这些警告消息；如果这个变量的值是 1（默认值），记载它们。如果这个变量的值大于 1，则加大记录级别把关于连接失败的信息和（从 MySQL 5.2.6 版开始）“拒绝访问”错误也包括在记载范围内。
- long_query_time（启动：直接设置；运行时：全局、会话）
  一个以秒为单位的时间值，如果某个查询命令的执行时间大于这个值，MySQL 服务器就将认为它是一个“慢”查询。每出现一个慢查询，Slow_queries 计数器变量的值就会加 1；此外，如果慢查询日志机制已启用，这个查询就将被记录到相应的日志里去。(从 MySQL 5.1.21 版开始，慢查询日志功能还会把 min_examined_row_limit 变量考虑在内。)
  这个变量的默认值是 10。从 MySQL 5.1.21/6.0.4 版开始，这个值还可以包括一个精确到毫秒的小数部分，而且最小值是 0。(只有当日志信息的输出目的地是一个文件而不是 mysql.slow_query 数据表的时候才能记录小数部分。)在更早的版本里，这个值必须是一个

整数，而且最小值是 1。

- low_priority_updates（启动：直接设置；运行时：全局、会话）

  当这个变量被设置为“真”的时候，对使用数据表级锁定机制的存储引擎来说数据更新操作将比数据检索操作的优先级低。对数据表内容进行修改的语句（DELETE、INSERT、REPLACE、UPDATE）将一直等到数据表上没有 SELECT 语句正在执行或等待执行的时候才开始执行。如果在一条 SELECT 语句尚未执行完毕的时候又来了一条 SELECT 语句，后者将在前者执行完毕后立刻开始执行，而不是排在低优先级的数据修改语句的后面等待。把 LOW_PRIORITY 选项添加到支持这个选项的语句（比如 INSERT 和 UPDATE 语句）里也可以获得同样的效果。对于各条 INSERT 语句而言，可以通过给它加上 HIGH_PRIORITY 选项的办法来抑制这个变量的效力，把该语句将要执行的数据插入操作提升到正常的优先级。

  low_priority_updates 变量还有一个目前已不常用的同义变量 sql_low_priority_updates。

- lower_case_file_system

  这个变量用来表明包含着MySQL数据目录的文件系统对文件名是否区分大小写。如果这个变量的值是ON，文件名是不区分大小写的（同一个文件名的大写和小写形式没有区别）；如果这个变量的值是OFF，文件名是区分大小写的。

- lower_case_table_names（启动：直接设置）

  这个变量控制着 MySQL 服务器在执行 CREATE DATABASE 和 CREATE TABLE 语句的时候将如何把数据库名和数据表名转换为相应的子目录名和文件名。在执行某些语句的过程中进行的名字比较操作也会受到这个变量的影响。

  - 如果这个变量的值是0，MySQL服务器将按照CREATE DATABASE和CREATE TABLE语句里给出的名字来创建磁盘文件。名字的比较操作将是区分大小写的。如果操作系统的文件名区分大小写，这将是默认设置。
  - 如果这个变量的值是1，MySQL服务器在创建数据库和数据表时将统一使用由小写字母构成的名字。名字的比较操作是区分大小写的。
  - 如果这个变量的值是2，MySQL服务器将按照CREATE DATABASE和CREATE TABLE语句里给出的名字来创建磁盘文件，但名字的比较操作不区分大小写。你应该只在文件名不区分大小写的系统上才使用这个值。

  在你没有明确地对 lower_case_table_names 变量做出设置的情况下，如果 MySQL 数据目录所在的文件系统对文件名不区分大小写，MySQL 服务器将自动地把这个变量设置为 2。把这个变量设置为一个非零值还将导致数据表的别名不区分大小写。

- max_allowed_packet（启动：直接设置；运行时：全局、会话）

  MySQL 服务器与客户通信时使用的缓冲区的最大长度。这个缓冲区的初始长度是 net_buffer_length 个字节，但在必要时可以增大到 max_allowed_packet 个字节。这个值对 MySQL 服务器所能处理的字符串的最大长度也有限制作用。这个变量的默认值和最大值分别是 1MB 和 1GB。

- max_binlog_cache_size（启动：直接设置；运行时：全局）

  二进制日志缓存的最大长度。在事务被提交之前，构成事务的语句都保存在二进制日志缓存里，它们要等到事务被提交时才会被写入二进制日志。如果某个事务超出了这个变量所设定

的最大长度，就必须被写入一个临时硬盘文件。

- max_binlog_size（启动：直接设置；运行时：全局）
  二进制日志文件的最大长度。如果当前的二进制日志文件达到了这个长度，MySQL 服务器将关闭它并开始使用下一个二进制日志文件。这个变量的取值范围是从 4KB 到 1GB，默认值是 1GB。
  如果 max_relay_log_size 变量被设置为 0，从服务器里的中继日志文件的最大长度也将由 max_binlog_size 变量控制。
- max_connect_errors（启动：直接设置；运行时：全局）
  如果某个主机在尝试连接 MySQL 服务器 max_connect_errors 次之后还没有成功，MySQL 服务器将阻塞来自那台主机的后续连接请求。这是为了防止有人从那台主机试图闯入数据库系统。如果想让被阻塞的主机可以继续尝试连接，可以使用 FLUSH HOSTS 语句或者 mysqladmin flush-hosts 命令去清除主机缓存。
- max_connections（启动：直接设置；运行时：全局）
  允许同时保持在打开状态的客户连接的最大个数。从 MySQL 5.1.15 版开始，这个变量的默认值是 151；在更早的版本里，这个变量的默认值是 100。
- max_delayed_threads（启动：直接设置；运行时：全局、会话）
  为处理 INSERT DELAYED 语句而允许创建的线程的最大个数。如果已经创建了这么多个线程又有新的 INSERT DELAYED 语句到来，新到的 INSERT DELAYED 语句将被当做非 DELAYED 语句来处理。每一个客户都可以通过把这个变量的会话值设置为 0 来为它自己的连接禁用 DELAYED 插入。
- max_error_count（启动：直接设置；运行时：全局、会话）
  MySQL 服务器将保存的出错消息、警告消息和提醒消息的最大条数。（这个变量只影响 MySQL 服务器将把多少条上述消息保存下来供 SHOW ERRORS 和 SHOW WARNINGS 语句显示，不影响它对这类消息/事件的计数。）
- max_heap_table_size（启动：直接设置；运行时：全局、会话）
  新建 MEMORY 数据表的最大长度。改变这个变量的值不影响那些已经存在的数据表，除非你在改变了这个变量的值以后又用 ALTER TABLE 或者 TRUNCATE TABLE 语句修改了它们。这个变量可以用来帮助预防 MySQL 服务器消耗过多的内存。这个变量还会影响到 MySQL 服务器将如何对待它内部使用的各种内存型数据表。请参见对 tmp_table_size 变量的描述。
- max_insert_delayed_threads（启动：使用-- max-delayed-threads选项；运行时：全局、会话）
  这个变量是 max_delayed_threads 变量的一个同义词。
- max_join_size（启动：直接设置；运行时：全局、会话）
  在对多个数据表进行联结时，MySQL 优化器会先对将被筛选的数据行的组合个数做一个估算。如果估算值超过了 max_join_size 个数据行，就直接返回一条出错信息。这可以有效地预防因出现用户编写的 SELECT 查询不合理而返回过量数据行的情况。由这个变量设定的上限值不适用于已经保存在查询缓存里的查询结果，因为缓存里的查询结果可以立刻返回而无需再次执行有关的查询命令。
  这个变量需要与 sql_big_selects 变量（它只有会话级形式）配合使用，详见对 sql_big_

selects 变量的描述。把 max_join_size 变量设置为一个不是 DEFAULT 的值将自动地把 sql_big_selects 变量设置为 0。

max_join_size 变量还有一个目前已不常用的同义变量 sql_max_join_size。

- max_length_for_sort_data（启动：直接设置；运行时：全局、会话）
  查询优化器将使用这个变量来判断应该为 ORDER BY 操作执行哪一种 filesort 动作。
- max_prepared_stmt_count（启动：直接设置；运行时：全局）
  MySQL 服务器可以同时处理的预处理语句的最大个数。这个变量的可取值是 0 到 1 000 000，默认值是 16 382。这个变量的值越小，MySQL 服务器占用内存就越少。这个变量是从 MySQL 5.0.21/5.1.10 版开始引入的。
- max_relay_log_size（启动：直接设置；运行时：全局）
  从服务器上的中继日志文件的最大长度。如果当前的中继日志文件达到了这个长度，从服务器将关闭它并开始使用下一个中继日志文件。如果这个变量的值是 0，从服务器将使用 max_binlog_size 变量来控制中继日志文件的最大长度。这个变量的非零值取值范围是从 4KB 到 1GB；默认值是 0。
- max_seeks_for_key（启动：直接设置；运行时：全局、会话）
  查询优化器在进行基于键的查找时会用到这个变量。如果某个索引的差异度比较低（独一无二的值比较少），优化器会认为进行键查找需要多次匹配，所以会进行全表扫描。这个变量的作用是让优化器 max_seeks_for_key 次匹配就可以命中，所以把这个变量设置成一个比较小的值将使优化器认为查找一个键最多需要进行多次索引查找，因而倾向于选择使用索引而不是进行一次全表扫描。
- max_sort_length（启动：直接设置；运行时：全局、会话）
  MySQL 将使用 BLOB 或 TEXT 值的前 max_sort_length 个字节对它们进行排序。这个变量的默认值是 1 024。如果 BLOB 或 TEXT 值的那些字节是独一无二的话，把这个变量设置为小值可以在不损失精确度的情况下缩短比较时间。反之，如果这许多字节中的已排序值不是唯一的，加大这个变量的值可以获得更好的排序效果。
- max_sp_recursion_depth（启动：直接设置；运行时：全局、会话）
  存储过程的最大递归深度。这个上限值针对的是每一个存储过程，不是它们的递归深度的总和。这变量的默认值是 0（不允许递归），最大值是 255。这个变量是从 MySQL 5.0.17 版开始引入的。
- max_tmp_tables（启动：直接设置；运行时：全局、会话）
  每个客户能够同时打开的临时数据表的最大个数。这个变量目前尚未正式投入使用。
- max_user_connections（启动：直接设置；运行时：全局、会话）
  允许每个账户同时保有的客户连接的最大个数。这个变量的默认值是 0，其含义是“没有限制”。不管这个变量的值是多少，每个账户能够同时保有的客户连接的个数还要受限于 max_connections 变量。
  这个变量的会话级值从 MySQL 5.0.3 版才开始存在，而且是只读的。如果 mysql.user 数据表里与某个账户相对应的数据行有一个非零的 MAX_USER_CONNECTIONS 值，该账户的会话级 max_user_connections 变量将等于那个非零值；否则，将与全局级 max_user_connections 变量的值一样。

如果需要为某个特定的账户设置一个 max_user_connections 上限值,可以使用 GRANT 语句。

- max_write_lock_count(启动:直接设置;运行时:全局)
  在对某个数据表使用了 max_write_lock_count 个写锁定之后,MySQL 服务器将适当提升为该数据表申请一个读锁定的语句的优先级。
- min_examined_row_limit(启动:直接设置;运行时:全局、会话)
  一个查询至少需要检查 min_examined_row_limit 个数据行才有资格被记入“慢查询”日志。这个变量的默认值是 0。这个变量是从 MySQL 5.1.21 版开始引入的。
- myisam_block_size(启动:直接设置)
  MyISAM 数据表的索引块的单位长度。
- myisam_data_pointer_size(启动:直接设置;运行时:全局)
  MyISAM 索引文件里的数据行指针以字节计算的长度。这个变量的取值范围是 2 到 7。从 MySQL 5.0.6 版开始,默认值是 6;在更早的版本里,默认值是 4。
  具体到某个特定的数据表,这个指针长度还会受到 MAX_ROWS 数据表选项的影响。
- myisam_max_sort_file_size(启动:直接设置;运行时:全局)
  在 MyISAM 数据表上,因执行 REPAIR TABLE、ALTER TABLE 或 LAOD DATA 等语句而导致的索引重建工作,既可以使用一个临时文件去完成,也可以使用键缓存去完成。MySQL 将根据这个变量的值来决定使用哪一种方法:如果临时文件的估算长度大于这个值,MySQL 就会使用键缓存去重建各有关的索引。
- myisam_recover_options(启动:使用-- myisam-recover选项)
  MySQL 服务器在启动时使用的--myisam-recover 选项的值;这个选项设定了 MyISAM 数据表的自动修复模式。
- myisam_repair_threads(启动:直接设置;运行时:全局、会话)
  在修复操作过程中用来创建 MyISAM 数据表索引的线程的个数。(转储清除操作只适用于通过排序进行的修复,不适用于使用键缓存进行的修复。)这个变量的默认值是 1,意思是使用单个线程进行修复。把这个变量设置为大于 1 的值将使用多个线程进行修复——但这目前还属于一种试验性的做法。
- myisam_sort_buffer_size(启动:直接设置;运行时:全局、会话)
  在 ALTER TABLE、CREATE INDEX 和 REPAIR TABLE 等操作期间,MySQL 为了对 MyISAM 数据表的索引进行排序而分配的缓冲区的长度。
- myisam_stats_method(启动:直接设置;运行时:全局、会话)
  这个变量控制着 MySQL 服务器在为 MyISAM 数据表统计其索引键的分布概率时是把 NULL 值视为彼此相同还是不同。这个变量的可取值是 null_equal(把所有 NULL 值归为一组)和 null_unequal(每个 NULL 值自成一组)。这个变量是从 MySQL 5.0.14 版开始引入的。在那之前,这种统计计算是按照与 null_equal 相同的方式进行的。
- myisam_use_mmap(启动:使用--myisam_use_mmap选项;运行时:全局)
  这个变量的值是 ON 或 OFF(默认值),决定着 MySQL 服务器是否使用内存映射来读写 MyISAM 数据表。这个变量是从 MySQL 5.1.4 版开始引入的。
- named_pipe(启动:使用--enable-named-pipe选项)
  对命名管道连接的支持机制是否已用。这种连接只能在基于 Windows 的系统上使用。

- net_buffer_length（启动：直接设置；运行时：全局、会话）
  MySQL 服务器与客户程序进行通信时使用的连接和结果缓冲区的初始长度。这个缓冲区可以扩充到 max_allowed_packet 个字节长。这个变量的取值范围是从 1KB 到 1MB；默认值是 16KB。
- net_read_timeout（启动：直接设置；运行时：全局、会话）
  在 MySQL 服务器接收客户数据的场合，如果 MySQL 服务器在等待了 net_read_timeout 秒之后仍未收到来自客户连接的数据，就将产生一个读操作超时错误。这种倒计时只适用于 TCP/IP 连接。
- net_retry_count（启动：直接设置；运行时：全局、会话）
  在 MySQL 服务器接收客户数据的场合，如果读操作因某种原因而中断，MySQL 服务器将重试该操作 net_retry_count 次数。
- net_write_count（启动：直接设置；运行时：全局、会话）
  在 MySQL 服务器往客户发送数据的场合，如果 MySQL 服务器在经过了 net_write_timeout 秒之后仍未收到来自客户的响应，就将产生一个写操作超时错误。这种倒计时只适用于 TCP/IP 连接。
- new（启动：直接设置；运行时：全局、会话）
  MySQL 4.0 版使用这个变量来控制 MySQL 服务器是否启用某些 4.1 版功能。它现在已经没有用了。
- old（启动：直接设置）
  这个用来解决版本兼容问题的选项可以让某些操作按“老”办法进行。它目前的作用是让 ORDER BY 或 GROUP BY 子句在执行时不使用索引作为提示。这个变量是从 MySQL 5.1.18 版开始引入的。
- old_password（启动：直接设置；运行时：全局、会话）
  从 MySQL 4.1 版开始，MySQL 服务器默认使用一种新的口令加密算法来验证用户的身份。4.1 版以后的 MySQL 服务器将根据这个变量的值来决定是否还要使用 4.1 版之前的口令加密算法。
- open_files_limit（启动：直接设置）
  这个变量的值是 MySQL 服务器试图保留的文件描述符的个数。如果你在启动 MySQL 服务器时把这个变量设置为一个非零值，但在 MySQL 服务器启动后发现它的实际值小于你给出的设置值，那个实际值将代表着操作系统允许 MySQL 服务器保留的文件描述符的最大个数。（如果这个变量的实际值是零，其含义是操作系统不允许 mysqld 程序改变文件描述符的个数。）如果你在启动 MySQL 服务器时没有设置这个变量或是把它设置为 0，MySQL 服务器将以 max_connections × 5 和 max_connections + table_cache × 2 这两者当中较大的那个数值作为它将保留的文件描述符的个数。在 open_files_limit 变量控制下分配的文件描述符与在 innodb_open_files 变量控制下分配的文件描述符互不相干。
- optimizer_prune_level（启动：直接设置；运行时：全局、会话）
  查询优化器将分析多个执行计划并从中选出一个最佳的。这个变量决定着优化器如何处理候选执行计划。如果这个变量的值是 1（默认值），优化器将只对各候选计划将要检查的数据行的个数进行估算并根据估算结果丢弃“坏”计划。如果这个变量被设置为 0，优化器将对每一

个候选计划做彻底的搜索。这个变量是从 MySQL 5.0.1 版开始引入的。

- optimizer_search_depth（启动：直接设置；运行时：全局、会话）
  这个变量控制着查询优化器搜索执行计划的深度。如果这个变量的值是 0，优化器将自动选择一个合理的深度值。默认行为是沿用 MySQL 5.0 版之前的做法，即进行彻底的搜索。这个变量是从 MySQL 5.0.1 版开始引入的。
- pid_file（启动：直接设置）
  这个变量给出的是一个文件路径名，MySQL 服务器将把自己的进程 ID 编号写到这个文件里去。
- plugin_dir（启动：直接设置）
  用来保存各种插件的子目录的路径名。这个变量是从 MySQL 5.1.2 版开始引入的。
- port（启动：直接设置）
  MySQL 服务器用来监听客户连接的 TCP/IP 端口号。
- preload_buffer_size（启动：直接设置；运行时：全局、会话）
  这个变量决定着 MySQL 服务器在使用 LOAD INDEX 语句预加载有关索引时将分配一个多大的缓冲区。
- protocol_version
  MySQL服务器所使用的客户/服务器协议的版本号。
- pseudo_thread_id
  这个变量目前仅供MySQL服务器内部使用。
- query_alloc_block_size（启动：直接设置；运行时：全局、会话）
  为了分析和执行 SQL 语句而分配的临时内存的块长度。
- query_cache_limit（启动：直接设置；运行时：全局）
  查询结果的最大缓存长度，超过这一长度的查询结果将不会被缓存。这个变量的默认值是 1MB。
- query_cache_min_res_unit（启动：直接设置；运行时：全局）
  为了把查询结果存入查询缓存而分配的内存的块长度。这个变量的默认值是 4KB。
- query_cache_size（启动：直接设置；运行时：全局）
  用来缓存查询结果的内存的长度。把这个变量设置为 0 将强行禁用查询缓存功能，即使 query_cache_type 变量的值不是 OFF 也会如此。反过来说，把这个变量设置为一个非零值将强行分配那么多的内存，即使 query_cache_type 变量的值是 OFF 也会如此。这个变量的值必须是 1 024 的整数倍。
- query_cache_type（启动：直接设置；运行时：全局、会话）
  查询缓存的操作模式，但它们只在 query_cache_size 变量的值大于 0 的时候才有实际意义。表 D-2 列出了允许使用的操作模式。

表 D-2

模　式	含　义
0	既不缓存查询结果，也不检索被缓存的结果
1	对可以缓存的查询命令进行缓存，但以SELECT SQL_NO_CACHE开头的查询命令不包括在内
2	只对以SELECT SQL_CACHE开头的可以缓存的查询命令进行缓存

如果使用 SET 语句来设置 query_cache_type 变量，可以使用符号值 OFF、ON 和 DEMAND 分别作为模式 0、模式 1 和 2 的同义词。

query_cache_type 变量还有一个目前已不常用的同义变量 sql_query_cache_type。

- query_cache_wlock_invalidate（启动：直接设置；运行时：全局、会话）

  当这个变量是 0（默认值）的时候，即使某个客户对某个数据表进行了 WRITE 锁定，其他客户也能检索该数据表已被缓存的查询结果。如果把这个变量设置为 1，在某个客户对某个数据表进行了 WRITE 锁定期间，该数据表已被缓存的查询结果将变得不可用，其他客户必须等待该锁定被解除。
- query_prealloc_size（启动：直接设置；运行时：全局、会话）

  为了分析和执行 SQL 语句而分配的缓冲区的长度。与那些在 query_alloc_block_size 变量控制之下分配的临时内存块不同，这个缓冲区不会在前后两条语句之间被释放。
- range_alloc_block_size（启动：直接设置；运行时：全局、会话）

  在进行范围优化时分配的内存的块长度。
- read_buffer_size（启动：直接设置；运行时：全局、会话）

  对数据表做顺序扫描的线程所使用的缓存区的长度。如有必要，每个客户（程序）都能分配到一个这样的缓冲区。
- read_only（启动：直接设置；运行时：全局）

  这个变量控制着从服务器是否以只读方式来处理客户连接。在默认的情况下，read_only 变量的值是 OFF；此时，只要客户拥有必要的权限，从服务器就会允许客户对数据表进行修改。如果把这个变量设置为 ON，从服务器将只允许从主服务器那里接收到的语句和具备 SUPER 权限的客户发出的语句对数据表进行修改。

  从 MySQL 5.1.15 版开始，MySQL 服务器在使用 read_only 变量时又增加了一些限制：如果你明确地锁定了某个数据表或是有尚未执行完毕的事务，你将不能启用这个变量。如果你试图在其他客户锁定了某个数据表或是有尚未执行完毕的事务时激活这个变量，你的请求将被阻塞，直到锁定被解除或是事务执行完毕为止。在你的请求被阻塞期间，如果其他客户试图申请一个新的数据表级锁定或是试图开始一个新的事务，它们也将被阻塞。这些阻塞条件对 FLUSH TABLES WITH READ LOCK 语句没有作用，因为该语句申请的是一个全局级读锁定，不是一个数据表级锁定。
- read_rnd_buffer_size（启动：直接设置；运行时：全局、会话）

  在对数据行进行排序之后，用来按顺序读取各数据行的缓冲区的长度。如有必要，每个客户（程序）都能分配到一个这样的缓冲区。
- relay_log_purge（启动：直接设置；运行时：全局）

  如果这个变量被设置为 1（默认值），从服务器就会在用完一个中继日志文件之后立刻删除它。如果把这个变量设置为 0，中继日志文件将不会被自动删除。
- relay_log_space_limit（启动：直接设置）

  全体中继日志文件的总长度的上限。
- rpl_recovery_rank（运行时：全局）

  这个变量目前尚未投入使用。
- secure_auth（启动：直接设置；运行时：全局）

如果这个变量被设置为 ON，MySQL 服务器将只允许那些使用新口令格式（从 MySQL 4.1 版开始启用）的账户来建立连接。如果这个变量被设置为 OFF，MySQL 服务器将允许那些使用老口令格式的账户也来建立连接。这个变量的默认值是 OFF。

- secure_file_priv（启动：直接设置）
  如果把这个变量设置为某个子目录的路径名，MySQL 服务器将只接受对该子目录里的文件进行操作的 LOAD DATA 和 SELECT ... INTO OUTFILE 语句以及 LOAD_FILE()函数。在默认的情况下，这个变量的值为空（没有上述限制）。这个变量是从 MySQL 5.0.38/5.1.17 版开始引入的。
- server_id（启动：直接设置；运行时：全局）
  MySQL 服务器的复制（replication）ID 编号。如果是 0，则表示该服务器不参与复制（replication）机制；否则，这个值必须是 1 到 $2^{32}-1$ 之间的一个整数。每一个服务器的复制 ID 编号都必须是独一无二的。
- shared_memory（启动：直接设置）
  如果这个变量被设置为 ON，MySQL 服务器将允许客户使用共享内存来建立连接。默认值是 OFF。共享内存连接目前只能在 Windows 系统上使用。
- shared_memory_base_name（启动：直接设置）
  用来建立共享内存连接的共享内存的名字。默认使用的名字是 MYSQL（区分大小写）。
- skip_external_locking（启动：直接设置）
  外部锁定机制（即文件系统级的锁定机制）是否被抑制。
- skip_networking（启动：使用-- skip-networking选项）
  OFF 表示允许 TCP/IP 连接，ON 表示禁用 TCP/IP 连接。在后一种场合，客户（程序）将只能从本地主机使用套接字（如果是 Unix 系统）或使用命名管道或共享内存（如果是 Windows 系统）进行连接。
- skip_show_databases（启动：直接设置）
  当这个变量被设置为 OFF（默认值）的时候，任何用户都可以使用 SHOW DATABASES 语句。具备 SHOW DATABASE 权限的用户可以查看所有的数据库，不具备 SHOW DATABASE 权限的用户只能查看自己有权访问的那些数据库。当这个变量被设置为 ON 的时候，只有具备 SHOW DATABASE 权限的用户才能使用 SHOW DATABASES 语句，但还是可以查看所有的数据库。
- slave_allow_batching（启动：直接设置；运行时：全局）
  启用从服务器的批请求功能，仅适用于 MySQL Cluster。这个变量是从 MySQL 5.2.5 版开始引入的。
- slave_compressed_protocol（启动：直接设置；运行时：全局）
  这个变量决定着是否要对从服务器和主服务器之间的通信进行压缩。这还需要主从服务器都支持使用压缩协议才行。
- slave_load_tmpdir（启动：直接设置）
  子目录路径名，从服务器将在其中为 LOAD DATA 语句创建临时文件。这个变量的默认值是 tmpdir 系统变量的值。
- slave_net_timeout（启动：直接设置；运行时：全局）
  如果从服务器在经过 slave_net_timeout 秒之后仍未接收到来自主服务器的数据，就将产生一个超时错误。这种倒计时仅适用于 TCP/IP 连接。

- slave_skip_errors（启动：使用-- slave-skip-errors选项）
  这个变量用来给出一列出错代码，如果在执行过程中发生了这个列表里列出的错误，从服务器将忽略之而不是挂起复制处理进程。（不过，与利用这个选项来忽略错误的做法相比，还是找出问题的根源并彻底解决更好。）如果这个变量的值是 all，则将忽略所有的错误；否则，这个变量的值应该是以逗号分隔的一个或者多个出错代码。
- slave_transaction_retries（启动：直接设置；运行时：全局）
  如果因为发生了死锁现象或是超出了与存储引擎有关的倒计时设置而导致某个事务执行失败，从服务器将重试 slave_transaction_retries 次之后才报告出错。这个变量是从 MySQL 5.0.3 版开始引入的。
- slow_launch_time（启动：直接设置；运行时：全局）
  如果某个线程用了多于 slow_launch_time 秒的时间才被创建出来，它就会被认为是一个“慢创建”线程，并将导致状态计数器 Slow_launch_threads 加 1。
- slow_query_log（启动：直接设置；运行时：全局）
  是否启用“慢查询”日志功能。（如果启用，日志信息的输出目的地将由 log_output 变量决定。）这个变量是从 MySQL 5.1.12 版开始引入的，但从 MySQL 5.1.23 版开始才允许在运行时设置。
- slow_query_log_file（运行时：全局）
  “慢查询”日志文件的名字，这个变量只在相关日志信息的输出目的地是文件的时候才起作用。这个变量是从 MySQL 5.1.12 版开始引入的。
- socket（启动：直接设置）
  Unix 域套接字的路径名或 Windows 系统中的命名管道的名字。
- sort_buffer_size（启动：直接设置；运行时：全局、会话）
  供那些用来完成排序操作（GROUP BY 或 ORDER BY）的线程使用的缓冲区的长度。如有必要，每个客户（程序）都能分配到一个这样的缓冲区。一般来说，如果你有许多个客户（程序）会在同一时间进行排序操作，就不应该把这个值设置得很大（超过 1MB）。
- sql_mode（启动：直接设置；运行时：全局、会话）
  MySQL 服务器的 SQL 模式。这个选项将改变 MySQL 服务器的某些行为，使它更符合 SQL 语言标准或是与其他数据库服务器或老版本的 MySQL 服务器保持兼容。这个变量的值应该是一个空字符串（这将清除以前设置的 SQL 模式）或者是由下面将要介绍的一个或多个模式值以逗号分隔而构成的一系列值。有些模式值很简单，它们可以单独使用以启用某种特定的行为。另有一些模式值对应着各种复合 SQL 模式，每种复合 SQL 模式涵盖多种简单 SQL 模式，这使得用户可以方便地一次设置多种 SQL 模式。模式值不区分大小写。
  在下面介绍 SQL 模式的时候，术语“严格模式”的含义是 sql_mode 变量值至少包含 STRICT_ALL_TABLES 和 STRICT_TRANS_TABLES 二者之一，而这将导致 MySQL 服务器对输入数据的合法性进行更严格的检查。3.3 节对严格模式和另外几个影响输入数据检查工作的 SQL 模式进行了详细的讨论。
  - ALLOW_INVALID_DATES
    在严格模式里，抑制对 DATE 和 DATETIME 值进行全面的日期合法性检查。唯一的要求是月份值必须在 1 到 12 之间，日期值必须在 1 到 31 之间。但 TIMASTAMP 值是个例外：不管是

否启用了这个 SQL 模式，它们都必须是合法的。

这个模式是从 MySQL 5.0.2 版开始引入的。在 5.0.2 版之前，MySQL 服务器一直按照已启用这个模式的方式来检查日期值。也就是说，早期的 MySQL 服务器对日期值的检查不那么严格。

- ANSI_QUOTES

  把双引号字符（"）解释为供标识符（比如数据库、数据表、数据列的名字）使用的引号字符而不是供字符串使用的引号字符。不管这个模式是否已启用，反引号（`）总是可以用做各种名字的引号字符。

- ERROR_FOR_DIVISION_BY_ZERO

  对于数据行插入或修改操作，即使是在严格模式下，以零为除数的除法（或求余数）运算通常会以 NULL 值作为结果且不返回任何警告消息。启用 ERROR_FOR_DIVISION_BY_ZERO 模式将改变这种行为。如果没有启用严格模式，以零为除数时结果将为 NULL 值，但会返回一条警告消息；如果已经启用了严格模式，在执行 INSERT 或 UPDATE 语句期间遇到以零为除数的情况时将产生一个错误并导致“肇事”语句执行失败。在 ERROR_FOR_DIVISION_BY_ZERO 模式已启用的前提下，INSERT IGNORE 或者 UPDATE IGNORE 语句可以抑制以零为除数时的错误消息并提供一个 NULL 值作为除法运算的结果，但仍将产生一条警告消息。这个模式是从 MySQL 5.0.2 版开始引入的。

- HIGH_NOT_PRECEDENCE

  这个模式是从 MySQL 5.0.2 版开始引入的。它将把 NOT 操作符的优先级提升到与“!”操作符相同；也就是 NOT 操作符在 MySQL 5.0.2 之前的版本里具备的优先级。

- IGNORE_SPACE

  如果启用了这个模式，MySQL 服务器将忽略内建函数的名字与其参数表开头的左括号之间的空格；否则，那个左括号必须紧跟在函数名的后面，它们之间不能有任何空格。启用这个模式将导致 MySQL 服务器把函数名当做保留字。

- NO_AUTO_CREATE_USER

  防止 GRANT 语句创建不安全的新账户。如果没有用 IDENTIFIED BY 子句正确地给出一个现有账户的口令，GRANT 语句将执行失败。这个模式是从 MySQL 5.0.2 版开始引入的。

- NO_AUTO_VALUE_ON_ZERO

  如果没有启用这个模式，把零值插入一个 AUTO_INCREMENT 数据列将有着与插入 NULL 值同样的效果：MySQL 将自动生成下一个序列编号并把它保存到那个数据列里。如果启用了这个模式，往一个 AUTO_INCREMENT 数据列里插入零值的效果将是实实在在地把数值 0 存入那个数据列。

- NO_BACKSLASH_ESCAPES

  如果启用了这个模式，MySQL 服务器将把反斜线字符（“\”）解释为一个没有特殊含义的普通字符，而不是一个转义序列引导字符。这个模式是从 MySQL 5.0.1 版开始引入的。

- NO_DIR_IN_CREATE

  忽略 CREATE TABLE 和 ALTER TABLE 语句里的 DATA DIRECTORY 和 INDEX DIRECTORY 数据表选项。

- NO_ENGINE_SUBSTITUTION

这个模式决定着 MySQL 服务器在遇到带有 ENGINE 选项的 CREATE TABLE 或 ALTER TABLE 语句但该选项所指定的存储引擎不可用时将如何处理它们。(从 MySQL 5.1.12 版开始，“不可用”的含义是“没有在编译时包括或无法在运行时加载”。在 5.1.12 版之前，它的含义是“没有在编译时包括，或者虽然在编译时包括但在运行时被禁用”。)

如果启用了这个模式，当指定的存储引擎不可用时，数据表将不会被创建（或被更改）并发生一个错误。如果禁用了这个模式，当指定的存储引擎不可用时，MySQL 将使用默认的存储引擎。在后一种情况下，只有当指定存储引擎的名字非法时才会发生一个错误；但这仅适用于 MySQL 5.1.2 之前的版本，因为只有 5.1.2 版之前的服务器才能在事先获得一份合法的存储引擎的名单。(在 MySQL 5.1.2 之后的版本里，因为存储引擎可以在运行时加载，MySQL 服务器将无法提前获得一份合法的存储引擎的名单。) 这个模式是从 MySQL 5.0.8 版开始引入的。

■ NO_FIELD_OPTIONS

让 SHOW CREATE TABLE 语句的输出不包含 MySQL 独有的与数据列有关的选项，其目的是为了让这条语句的输出更好的可移植性。

■ NO_KEY_OPTION

让 SHOW CREATE TABLE 语句的输出不包含 MySQL 独有的与索引有关的选项，其目的是为了让这条语句的输出有更好的可移植性。

■ NO_TABLE_OPTIONS

让 SHOW CREATE TABLE 语句的输出不包含 MySQL 独有的与数据表有关的选项，其目的是为了让这条语句的输出有更好的可移植性。

■ NO_UNSIGNED_SUBTRACTION

在默认的情况下，如果在参与整数减法运算的两个操作数里有一个是无符号整数，计算结果就将是一个无符号整数。启用这个模式将使上述运算的结果是一个带符号整数，这与 MySQL 4.0 版之前的行为保持兼容。

■ NO_ZERO_DATE

在严格模式下，拒绝接受'0000-00-00'作为一个合法日期值。在普通情况下，MySQL 允许存储“零”日期值。这个模式可以通过使用 INSERT IGNORE 语句代替 INSERT 语句的办法来覆盖。这个模式是从 MySQL 5.0.2 版开始引入的。

■ NO_ZERO_IN_DATE

在严格模式下，拒绝接受月份或天数是零的日期值（年份是零的日期值是允许的）。在普通情况下，MySQL 允许存储这样的日期值。在非严格模式下或如果用户发出的是 INSERT IGNORE 语句，MySQL 将把这样的日期值保存为'0000-00-00'。这个模式是从 MySQL 5.0.2 版开始引入的。

■ ONLY_FULL_GROUP_BY

在普通情况下，MySQL 不要求 SELECT 语句的结果集或是 HAVING 子句（如果有的话）里的非总计型数据列必须出现在 GROUP BY 子句（如果有的话）里，如下所示：

```
SELECT a, b, COUNT(*) FROM t GROUP BY a;
```

如果启用了 ONLY_FULL_GROOUP_BY 模式，SELECT 语句的结果集或是 HAVING 子句（如果有的话）里的非总计型数据列就必须出现在 GROUP BY 子句（如果有的话）里，如下所示：

```
SELECT a, b, COUNT(*) FROM t GROUP BY a, b;
```

- PAD_CHAR_TO_FULL_LENGTH

  在普通情况下，MySQL 服务器检索 CHAR 数据列值时会删除其尾缀空格。如果启用了这个模式，MySQL 服务器将保留 CHAR 数据列值的尾缀空格，让它们的长度等于数据列的定义宽度。这个模式是从 MySQL 5.1.20 版开始引入的。

- PIPES_AS_CONCAT

  如果启用了这个模式，MySQL 服务器将把“||”解释为字符串合并操作符而不是逻辑 OR 操作符。

- REAL_AS_FLOAT

  如果启用了这个模式，REAL 数据类型将成为 FLOAT 的同义词而不是 DOUBLE 的同义词。

- STRICT_ALL_TABLES

  如果启用了这个模式，所有的存储引擎都将对输入数据做更严格的检查，这将导致 MySQL 拒绝接受绝大多数非法值。这个模式是从 MySQL 5.0.2 版开始引入的。TRADITIONAL 模式比这个模式还要严格。

- STRICT_TRANS_TABLES

  如果启用了这个模式，事务型存储引擎将对输入数据做更严格的检查，这将导致 MySQL 拒绝接受绝大多数非法值。在此基础上，只要有可能（比如说，在遇到一条插入单个数据行的 INSERT 语句时），非事务型存储引擎也将对输入数据做更严格的检查。这个模式是从 MySQL 5.0.2 版开始引入的。TRADITIONAL 模式比这个模式还要严格。

表 D-3 列出了 MySQL 目前支持的复合 SQL 模式以及每种复合模式所涵盖的简单模式。

表　D-3

复合模式	所涵盖的简单模式
ANSI	ANSI_QUOTES, IGNORE_SPACE, PIPES_AS_CONCAT, REAL_AS_FLOAT
DB2	ANSI_QUOTES, IGNORE_SPACE, NO_FIELD_OPTIONS, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, PIPES_AS_CONCAT
MAXDB	ANSI_QUOTES, IGNORE_SPACE, NO_AUTO_CREATE_USER, NO_FIELD_OPTIONS, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, PIPES_AS_CONCAT
MSSQL	ANSI_QUOTES, IGNORE_SPACE, NO_FIELD_OPTIONS, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, PIPES_AS_CONCAT
MYSQL323	HIGH_NOT_PRECEDENCE, NO_FIELD_OPTIONS
MYSQL40	HIGH_NOT_PRECEDENCE, NO_FIELD_OPTIONS
ORACLE	ANSI_QUOTES, IGNORE_SPACE, NO_AUTO_CREATE_USER, NO_FIELD_OPTIONS, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, PIPES_AS_CONCAT
POSTGRESQL	ANSI_QUOTES, IGNORE_SPACE, NO_FIELD_OPTIONS, NO_KEY_OPTIONS, NO_TABLE_OPTIONS, PIPES_AS_CONCAT
TRADITIONAL	ERROR_FOR_DIVISION_BY_ZERO, NO_AUTO_CREATE_USER, NO_ZERO_DATE, NO_ZERO_IN_DATE, STRICT_ALL_TABLES, STRICT_TRANS_TABLES

在 MySQL 5.0.3 之前的版本里，ANSI 复合模式还包括 ONLY_FULL_GROUP_BY。

TRADITIONAL 复合模式之所以会有这样一个名字，是因为它启用的各项模式将导致 MySQL 服务器在对输入数据进行检查时表现得像是一个拒绝接受非法数据的传统的数据库系统，它在严格模式的基础上增加了一些更严格的限制。TRADITIONAL 模式是从 MySQL 5.0.2 版开始引入的。

- sql_select_limit（运行时：全局、会话）

  这个变量用来设定一条 SELECT 语句所能返回的数据行的最大个数。如果明确地给出了 LIMIT

子句，该子句的优先级比这个变量更高。默认设置是每个数据表所能容纳的数据行的最大个数。如果你曾经改变过这个变量，可以通过把这个变量设置为 DEFAULT 来恢复其默认设置。这个变量对存储例程或不把数据行返回给客户的 SELECT 操作（如子查询、INSERT INTO ... SELECT 和 CREATE TABLE ... SELECT 等语句）没有任何作用。

- sql_slave_skip_counter（运行时：全局）
  具备 SUPER 权限的用户可以把它作为 GLOBAL 变量设置为 *n*，由此让复制机制中的从服务器跳过接下来将从其主服务器接收的 *n* 个事件。
- ssl_xxx（启动：使用--ssl-xxx选项）
  ssl_xxx 存放着在启动 MySQL 服务器时使用的各--ssl-xxx 选项的值。（比如说，ssl_ca 变量存放着--ssl-ca 选项的值。）如果你在启动 MySQL 服务器时没有给出某个--ssl-xxx 选项，相应的 ssl_xxx 变量的值将是一个空字符串。如果 SSL 支持不可用，这些变量的值都将是 NULL。这些变量是从 MySQL 5.0.23/5.1.11 版开始引入的。
- storage_engine（启动：使用--default-storage-engine选项；运行时：全局、会话）
  MySQL 服务器默认使用的存储引擎。如果用户在创建数据表的时候没有给出 ENGINE = engine_name 选项或者通过这个选项给出的 engine_name 值没有被支持，MySQL 服务器就将使用这个存储引擎。
- sync_binlog（启动：直接设置；运行时：全局）
  当这个变量被设置为 0（默认值）的时候，MySQL 服务器将不把二进制日志转储到磁盘。当这个变量被设置为一个正整数 *n* 的时候，服务器每对二进制日志进行 *n* 次写操作就会把日志转储到磁盘。这个变量的正整数值越小，有关数据在系统发生崩溃时的安全系数就越大，但这同时对系统性能的负面影响也越大。
- sync_frm（启动：直接设置；运行时：全局）
  当这个变量被设置为 0 的时候，MySQL 服务器在创建一个非临时数据表时将不会把它的.frm 文件转储到磁盘。这个变量的默认值是 1，也就是 MySQL 服务器会把.frm 文件转储到磁盘。
- system_time_zone
  MySQL服务器的系统时区。MySQL服务器在它启动时会尝试通过询问操作系统的办法来确定这个变量的值。我们可以通过设置TZ环境变量或是使用mysqld_safe脚本和它的--timezone 选项来启动MySQL服务器的办法明确地设置这个值。
- table_cache、table_open_cache（启动：直接设置；运行时：全局）
  能够同时打开的数据表的最大个数。这个缓存是由全体线程共享的。这个变量最初的名字是 table_cache，但从 MySQL 5.1.3 版开始被改为 table_open_cache。
- table_definition_cache（启动：直接设置；运行时：全局）
  MySQL 服务器可以在它的“定义”缓存里存放的数据表定义（来自.frm 文件）的最大个数。这个变量是从 MySQL 5.1.3 版开始引入的。
- table_lock_wait_timeout（启动：直接设置；运行时：全局）
  如果某个已打开游标的连接在等待了 table_lock_wait_timeout 秒之后还没有获得一个数据表级锁定，就将发生一个超时错误。默认值是 50。这个变量是从 MySQL 5.0.10 版开始引入的。
- table_type（启动：直接设置；运行时：全局、会话）
  这个变量是 storage_engine 变量的一个同义词，但目前已被淘汰，MySQL 5.2.5 版已彻底删

除了它。

- thread_cache_size（启动：直接设置；运行时：全局）
  线程缓存所能容纳的线程的最大个数。当某个客户断开与 MySQL 服务器的连接时，如果这个缓存还没有满，该客户曾经使用过的线程就将被放入这个缓存。只要这个缓存里还有线程可用，新建立的连接就会重复使用它们而不是去创建新的线程。当 MySQL 服务器为每个当前连接的客户分别使用一个线程的时候，就要用到线程缓存。
- thread_concurrency（启动：直接设置；运行时：全局）
  这个变量只适用于 Solaris 系统。这个值将被传递给 thr_concurrency()函数，Solaris 系统上的线程管理器将参照这个值来决定应该同时运行多少个线程。
- thread_handling（启动：直接设置）
  这个变量控制着 MySQL 服务器将使用哪一种线程模型来处理客户连接。这个变量的值可以是 one-thread（用一个线程来处理所有的客户连接请求）、one-pool-pool-connection（为每个当前连接的客户分别使用一个线程）或 pool-of-threads（用固定数量的线程池来处理所有的已连接客户，仅适用于 MySQL 6.0.4 以后的版本）。这个变量是从 MySQL 5.1.17 版开始引入的。
- thread_pool_size（启动：直接设置）
  如果 thread_handling 变量的值是 pool-of-threads，MySQL 服务器将为语句处理线程创建一个缓存池，这个变量的值就是那个缓存池所能容纳的线程的个数。它的默认值是 20。这个变量是从 MySQL 6.0.4 版开始引入的。
- thread_stack（启动：直接设置）
  每个线程的栈的长度。
- time_format
  这个变量目前尚未投入使用。
- time_zone（启动：使用--default-time-zone选项；运行时：全局、会话）
  MySQL 服务器的当前时区。如果这个变量的值是 SYSTEM，MySQL 服务器将使用 system_time_zone 变量的值作为它的当前时区。各个客户可以通过修改这个变量的会话级值为自己的连接另行设置一个时区。
- timed_mutexes（启动：使用--timed-mutexes选项；运行时：全局、会话）
  这个变量用来控制是否需要收集 InnoDB 互斥计时信息。这个变量是从 MySQL 5.0.3 版开始引入的。
- tmp_table_size（启动：直接设置；运行时：全局、会话）
  MySQL 内部使用的各种临时数据表（即 MySQL 服务器在处理 SQL 语句的过程中自动创建的数据表）以字节计算的最大允许长度。如果某个临时数据表的长度超过了 max_heap_table_size 和 tmp_table_size 两者当中比较小的那个值，MySQL 服务器就会把它转换为一个 MyISAM 数据表并保存到磁盘上去。如果你有足够多的内存的话，加大这个变量的值将使 MySQL 服务器能够在内存里维护更大的临时数据表而不必把它们转换为磁盘文件格式。
- tmpdir（启动：直接设置）
  MySQL 服务器将在其中创建临时文件的子目录的路径名。这个变量的值可以是一组目录路径名，MySQL 服务器将以轮转方式使用它们。在列出多个路径名的时候，在 Unix 系统上需要使用冒号（:）作为它们彼此之间的分隔符；在 Windows 或 NetWare 系统上需要使用分号（;）作

为分隔符。

- transaction_alloc_block_size（启动：直接设置；运行时：全局、会话）
  在提交过程中，在把某个事务写入二进制日志之前，为了处理构成该事务的语句而分配的临时内存的块长度。
- transaction_prealloc_size（启动：直接设置；运行时：全局、会话）
  为了处理构成某个事务的语句而分配的缓冲区的长度。与那些在 transaction_alloc_block_size 变量控制之下分配的临时内存块不同,这个缓冲区不会在前后两条语句之间被释放。
- tx_isolation（启动：使用--timed-isolation选项；运行时：全局、会话）
  MySQL 服务器默认使用的事务隔离级别。
- updatable_views_with_limit（启动：直接设置；运行时：全局、会话）
  当这个变量被设置为 0 或 OFF 的时候，如果某个视图更新操作（UPDATE 或 DELETE 语句）没有使用其底层数据表里的主键，MySQL 服务器将不允许它执行，哪怕使用了一条 LIMIT 1 子句把更新范围限制在单个数据行也是如此。当这个变量被设置为 1 或 YSE（默认值）的时候，MySQL 将允许上述更新操作执行，但会发出一条警告消息。这个变量是从 MySQL 5.0.2 版开始引入的。
- version
  MySQL服务器的版本。这个变量的值由一个版本编号以及（可能）一个或多个后缀构成。那些后缀值及其含义可以在附录C对VERSION()函数的描述里查到。
- version_comment
  在MySQL服务器的编译配置阶段利用configure的--with-comment选项给出的一条版本注释。如果你在配置阶段没有给出任何注释，这个变量的默认值将是Source distribution。
- version_compile_machine
  编译MySQL软件时使用的计算机硬件类型。这个值是在MySQL的配置阶段确定的。
- version_compile_os
  编译MySQL软件时使用的操作系统。这个值是在MySQL的编译配置阶段确定的。
- wait_timeout（启动：直接设置；运行时：全局、会话）
  如果某个非交互式的客户连接在 wait_timeout 秒内没有操作动作，MySQL 服务器就将认为该客户连接不再有保留的必要并自动关闭这个连接。对于交互式的客户连接，MySQL 服务器将使用 interactive_timeout 变量的值作为这种超时等待的秒数。这个变量仅适用于 TCP/IP 连接和 Unix 系统上的套接字文件连接。

D.2　只存在于会话级的系统变量

本节将要介绍的系统变量只存在于会话级。也就是说，连接到 MySQL 服务器的每一个客户都可以在当前会话里使用这些变量，但没有任何相应的全局级变量。个别客户对这些变量的设置只会影响到 MySQL 服务器对该客户本身的的操作。

这些只存在于会话级的系统变量中的绝大多数不会出现在 SHOW VARIABLES 语句的输出里，但只要知道了它们的名字，就可以利用 SELECT @@SEEION.*var_name* 或者 SELECT @@*var_name* 语句查出它们的值来。

会话变量的名字是不区分大小写的。

- autocommit
  用于事务处理的“自动提交”模式。这个变量的默认值是1，意思是启用自动提交模式，语句立刻生效。实际上，这相当于每条语句本身就是一个事务。把这个变量设置为0将禁用自动提交模式，随后的语句将一直等到执行了提交操作（通过发出一条COMMIT语句或者把autocommit变量重新设置为1）之后才会生效。只要某个事务还没有被提交，该事务里的语句就可以用ROLLBACK语句撤销掉。把autocommit变量设置回1将再次启用自动提交模式（并隐含地提交所有尚未被提交的事务）。
- big_tables
  如果把这个变量设置为1，MySQL内部使用的所有临时数据表都将被存储到磁盘上而不是驻留在内存里。这会降低MySQL服务器的性能，但可以让那些需要用到大临时数据表的SELECT语句不再因为“table full”（数据表满）错误而无法完成。这个变量的默认值是0（把临时数据表存储在内存里）。你通常用不着对这个变量进行设置。
  sql_big_tables 是 big_tables 的同义词，但已弃用。
- error_count
  这是一个只读变量，它的值是最近一条执行出错的语句所产生的错误的个数。
- foreign_key_checks
  把这个变量设置为0或1将禁用或者启用针对为InnoDB数据表的外键检查功能。这个变量的默认设置是执行这种检查。某些场合需要禁用外键检查功能，比如说，在恢复转储文件的过程中，可能各数据表的创建和加载顺序与外键关系所要求的顺序不一致，所以在开始加载数据表之前最好先把外键检查功能关掉，等数据表全部加载完毕后再重新启用。
- identify
  这个变量是last_insert_id会话变量的一个同义词。
- insert_id
  如果把这个变量设置为*n*，下一条INSERT语句往数据表里插入的AUTO_INCREMENT值就将是*n*。二进制日志的处理工作经常会用到这个变量。
- last_insert_id
  如果把这个变量设置为*n*，LATS_INSERT_ID()函数调用的返回值就将是*n*。二进制日志的处理工作经常会用到这个变量。
- sql_auto_is_null
  如果这个变量被设置为1（默认值），我们就可以用WHERE *col_name* IS NULL形式的WHERE子句来查知最近一次生成的AUTO_INCREMENT编号值，其中*col_name*就是那个AUTO_INCREMENT数据列的名字。有不少ODBC程序需要用到这一功能。把这个变量设置为0即可禁用上述功能。
- sql_big_selects
  这个变量经常与max_join_size系统变量配合使用。在sql_big_selects变量被设置为1（默认值）的时候，服务器允许查询命令返回任意大的结果集。在sql_big_selects变量被设置为0的时候，MySQL服务器将拒绝执行那些极有可能返回大量数据行的查询命令。此时，联结操作将受到max_join_size变量的约束：MySQL服务器先估算出需要对多少种数据行组合情况进行检查，如果这个估算值大于max_join_size变量的值，服务器就将返回一条出错信息

而不是去执行这个查询。

把 max_join_size 变量设置为一个不是 DEFAULT 的值将自动地把 sql_big_selects 变量设置为 0。

- sql_buffer_result

  把这个变量设置为1将导致MySQL服务器使用内部临时数据表来保存SELECT语句的查询结果。这么做的好处是可以让服务器尽快释放它在查询过程中锁定的各有关数据表。这个变量的默认值是0。

- sql_log_bin

  把这个变量设置为0或1将分别为当前客户连接禁用或启用二进制日志功能。客户必须具备SUPER权限才能让这条语句发挥作用。如果服务器的二进制日志功能未被启用的话，这个变量将没有任何效果。

- sql_log_off

  把这个变量设置为0或1将分别为当前客户连接启用或禁用把SQL语句记载到普通查询日志的功能。客户必须具备SUPER权限才能让这条语句发挥作用。如果服务器的普通查询日志功能未被启用的话，这个变量将没有任何效果。

- sql_log_update

  这个变量从MySQL 5.0版开始不再使用，因为改动日志已被去除。

- sql_notes

  把这个变量设置为0或1（默认值）将使MySQL服务器抑制或记录Note级别的警告消息。这个变量是从MySQL 5.0.3版开始引入的。

- sql_quote_show_create

  这个变量决定着是否需要把SHOW CREATE TABLE和SHOW CREATE DATABASE语句的输出里的标识符（即数据库、数据表、数据列和索引的名字）用引号括起来。它的默认值是1（使用引号）。如果你想把CREATE TABLE语句拿到一个不支持反单引号的数据库系统（如一些其他品牌的数据库服务器、非常老的MySQL服务器，等等）上使用的话，把这个变量设置为0（禁用反单引号）应该会有所帮助。请注意，如果真的禁用了这项功能，就一定要保证在数据表里使用的每一个名字既不是MySQL保留字也不包含特殊字符。

  如果没有启用 ANSI_QUOTE SQL 模式，标识符引号是反单引号字符（`）；如果启用了它，标识符引号是双引号（"）。

- sql_safe_updates

  如果这个变量被设置为1，MySQL服务器将只允许两种UPDATE和DELETE语句执行：(1) 将被修改的记录是通过键值确定的，(2) 使用了LIMIT子句。这个变量的默认值是0，即不实行上述限制。

- sql_warnings

  如果这个变量被设置为1，就算是一个单数据行插入操作，MySQL也会去统计和报告它的警告计数值。如果这个变量被设置为0（默认值），则只为插入多个数据行的INSERT语句统计和报告警告计数值。

- timestamp

  这个变量用来为当前连接设定一个TIMESTAMP值。二进制日志的处理工作会用到这个变量。

timestamp变量会影响NOW()函数的返回值，但不影响SYSDATE()函数的返回值。

- unique_checks
  把这个变量设置为0或1将禁用或者启用对InnoDB数据表里的次要索引进行的唯一性检查。禁用这些检查可以加快把数据导入InnoDB数据表的操作。反过来说，如果不能确定将被导入的数据是不是违反了InnoDB数据表的唯一性要求，千万不要禁用这些检查。
- warning_count
  这是一个只读变量，它的值是最近一条出了问题的语句所产生的错误、警告和提醒消息的总数。

D.3 状态变量

本节描述的状态变量可以提供各种关于 MySQL 服务器当前操作状态的信息。这些状态变量可以通过 SHOW STATUS 语句或通过执行 mysqladmin extended-status 命令的办法来查看。从 MySQL 5.0.2 版开始，状态变量也有了全局级和会话级之分（就像系统变量那样）：全局级状态变量的值与全体客户相关，会话级状态变量的值只与当前客户有关。如果某个状态变量只有一个全局级值，在全局级和会话级查询该变量将返回同样的值。从 MySQL 5.1.12 版开始，我们还可以通过查询 INFORMATION_SCHEMA 数据库中的 GLOBAL_STATUS 和 SESSION_STATUS 数据表的办法来获得状态变量信息。

关于如何在运行时查看状态变量的更多信息见 12.6.3 节。

状态变量的名字是不区分大小写的。

列在下面这份清单里的状态变量都是比较通用的。随后几个小节列出了一些彼此相关的状态变量，内容涉及以下几个方面：语句计数器、InnoDB 存储引擎、查询缓存和 SSL。

- Aborted_clients
  因客户没有正确地关闭而被丢弃的客户连接的个数。
- Aborted_connects
  试图连接MySQL服务器但没有成功的次数。
- Binlog_cache_disk_use
  因为尺寸超过了binlog_cache_size系统变量的值而不得不使用一个磁盘临时文件的事务的个数。
- Binlog_cache_use
  因为尺寸没有超过binlog_cache_size系统变量的值而保存在二进制日志缓存里的事务的个数。
- Bytes_received
  MySQL服务器从所有客户那里接收到的字节总数。
- Bytes_sent
  MySQL服务器向所有客户发送出去的字节总数。
- Com_*xxx*
  MySQL使用了一组状态变量来充当语句计数器，对曾经执行各类语句（命令）的次数进行统计。语句计数器状态变量的数量有几十个，名字都相似，这里就不把它们逐一地列举出来了。这些语句计数器状态变量的名字全都以Com_开头，都有一个表明某种具体语句类型的后缀。

比如说，Com_select和Com_drop_table变量将分别表明MySQL服务器曾经执行过多少条SELECT和DROP TABLE语句。

- Compression
  客户/服务器之间的通信是否使用了压缩。这个变量是从MySQL 5.0.16版开始引入的。
- Connections
  试图连接MySQL服务器的尝试次数（成功或失败的连接尝试都包括在内）。
- Created_tmp_disk_tables
  MySQL服务器在对SQL查询语句进行处理时在磁盘上创建的临时数据表的个数。
- Created_tmp_files
  MySQL服务器所创建的临时文件的个数。
- Created_tmp_tables
  MySQL服务器在对SQL查询语句进行处理时在内存里创建的临时数据表的个数。
- Delayed_errors
  在处理INSERT DELAYED数据行时发生的出错的个数。
- Delayed_insert_threads
  INSERT DELAYED处理线程的个数。
- Delayed_writes
  已被写入数据库的INSERT DELAYED数据行的个数。
- Flush_commands
  已执行完成的FLUSH语句的个数。
- Handler_commit
  提交一个事务的请求的个数。
- Handler_delete
  从数据表里删除一个数据行的请求的个数。
- Handler_discover
  这个变量是供NDBCLUSTER存储引擎使用的。它代表着MySQL服务器向NDB询问一个数据表的名字并成功找到该数据表的次数。
- Handler_prepare
  为两阶段提交操作进行准备的次数。这个变量是从MySQL 5.0.3版开始引入的。
- Handler_read_first
  读取索引中的第一个索引项的请求的个数。
- Handler_read_key
  根据一个索引值而读取一个数据行的请求的个数。
- Handler_read_next
  按索引顺序读取下一个数据行的请求的个数。
- Handler_read_prev
  按索引逆序读取前一个数据行的请求的个数。
- Handler_read_rnd
  根据某个数据行的位置而读取该数据行的请求的个数。

- Handler_read_rnd_next
  读取下一个数据行的请求的个数。如果这个数字很高，就说明有很多语句需要通过全表扫描才能完成或者有很多查询没有使用适当的索引。
- Handler_rollback
  回滚一个事务（即撤销该事务）的请求的个数。
- Handler_savepoint
  创建一个事务保存点的请求的个数。这个变量是从MySQL 5.0.3版开始引入的。
- Handler_savepoint_rollback
  回滚到一个事务保存点的请求的个数。这个变量是从MySQL 5.0.3版开始引入的。
- Handler_update
  对数据表里的一个数据行进行修改的请求的个数。
- Handler_write
  往数据表里插入一个数据行的请求的个数。
- Innodb_*xxx*
  与InnoDB有关的状态变量，详见D.3.1节。
- Key_blocks_not_flushed
  键缓存里已经被修改但还未被写入磁盘的缓存块的个数。
- Key_blocks_unused
  键缓存里尚未被使用过的缓存块的个数。
- Key_blocks_used
  键缓存里已被使用的缓存块的个数。
- Key_reads_requests
  从键缓存读取一个缓存块的请求的个数。
- Key_reads
  从磁盘读出索引块的读操作的次数。
- Key_write_requests
  向键缓存写一个缓存块的请求的个数。
- Key_writes
  把索引块写入磁盘的写操作的次数。
- Last_query_cost
  查询优化器最近一次的查询开销计算结果。这个值只对没有使用UNION或者子查询的查询才有用。如果还没有计算过查询开销，这个值将是0或-1（MySQL 5.0.7版之前）。从MySQL 5.0.16版开始，利用查询缓存完成的查询被纳入到这个变量的涵盖范围内。这个变量是从MySQL 5.0.1版开始引入的。
- Max_used_connections
  此前曾同时处于打开状态的连接的最大个数。
- Not_flushed_delayed_rows
  等待INSERT DELAYED语句写入的数据行的个数。
- Open_files

当前处于打开状态的文件的个数。

- Open_streams

  当前处于打开状态的流的个数。“流”是用fopen()函数打开的文件，这只适用于日志文件。

- Open_table_definitions

  被缓存在内存里的.frm文件的个数。这个变量是从MySQL 5.1.3版开始引入的。

- Open_tables

  当前处于打开状态的数据表的个数。不包括TEMPORARY表。

- Opened_files

  MySQL服务器已打开的文件的总数。（有几个存储引擎不影响这个计数器。）这个变量是从MySQL 5.1.21版开始引入的。

- Opened_tables

  MySQL服务器已打开的数据表的总数。如果这个数字很高，就应该考虑加大数据表缓存。

- Prepared_stmt_count

  预处理语句的个数。这个变量是在MySQL 5.0.21/5.1.10版作为prepared_stmt_count系统变量引入的，但从MySQL 5.0.32/5.1.11版开始被改成了Prepared_stmt_count状态变量。

- Qcache_*xxx*

  与查询缓存有关的状态变量，详见D.3.2节。

- Questions

  MySQL服务器已接收到的语句的个数（成功和不成功的都包括在内）。Questions和Uptime的比值就是MySQL服务器每秒接收到的查询个数。

- Rpl_status

  这个变量目前尚未投入使用。

- Select_full_join

  没有使用索引而完成的多数据表联结操作的次数。

- Select_full_range_join

  利用一个辅助性的参照表（reference table）上的区间搜索（range search）操作而完成的多数据表联结操作的次数。

- Select_range

  利用第一个数据表上的某个区间而完成的多数据表联结操作的次数。

- Select_range_check

  必须使用区间搜索（range search）操作才能从后续的数据表里检索数据行的多数据表联结操作的次数。

- Select_scan

  通过对第一个数据表进行全表扫描而完成的多数据表联结操作的次数。

- Slave_open_tmp_tables

  从服务器中的SQL线程曾经打开过的临时文件的个数。

- Slave_retried_transactions

  从服务器中的SQL线程重新尝试执行一个事务的次数。这个变量是从MySQL 5.0.4版开始引入的。

- Slave_running
  从服务器中的I/O线程和SQL线程是不是都在运行中。
- Slow_launch_threads
  花费了长于slow_lanuch_time秒的时间才创建出来的线程的个数。
- Slow_queries
  花费了长于long_query_time秒的时间才执行完毕的查询的个数。
- Sort_merge_passes
  排序算法进行的遍历次数。
- Sort_range
  利用一个区间进行的排序操作的次数。
- Sort_rows
  对多少行排了序。
- Sort_scan
  利用一次全表扫描操作而完成的排序操作的次数。
- Ssl_*xxx*
  与SSL有关的状态变量，详见D.3.3节。
- Table_locks_immediate
  无需等待就能够立刻得到满足的数据表锁定请求的个数。
- Table_locks_waited
  必须等待一段时间才能得到满足的数据表锁定请求的个数。如果这个数字很高，就表明数据表锁定存在问题。
- Tc_log_max_pages_used
  曾用于事务协调恢复日志文件的页面的最大个数。这个变量是从MySQL 5.0.3版开始引入的。
- Tc_log_page_size
  事务协调恢复日志文件的页面长度。这个变量是从MySQL 5.0.3版开始引入的。
- Tc_log_page_waits
  一个两阶段提交操作必须等到事务协调恢复日志文件里可用的页面时才能开始执行，这个状态变量记录了这种等待的次数。这个变量是从MySQL 5.0.3版开始引入的。
- Threads_cached
  线程缓存里现在有多少个线程。
- Threads_connected
  现正处于打开状态的连接的个数。
- Threads_created
  为处理客户连接而已创建的线程的总数。
- Threads_running
  现正处于活跃状态（非休眠状态）的线程的个数。
- Uptime
  MySQL服务器自开始运行以来已经持续运行的时间，以秒为计量单位。
- Uptime_since_flush_status

自从最近一次执行FLUSH STATUS语句以来的秒数。这个变量是从MySQL 5.0.35版开始引入的。这项功能目前只有MySQL Comminity Server发行版本具备。

D.3.1　与InnoDB有关的状态变量

下面列出的状态变量可以提供关于 InnoDB 存储引擎的操作状态的信息。它们当中的许多会出现在 SHOW ENGINE INNODB STATUS 语句的输出里，但在 SHOW STATUS 语句的输出里更容易解读。下列状态变量中的绝大多数都是从 MySQL 5.0.2 版开始引入的，例外情况将随时注明。

- Innodb_buffer_pool_pages_data
  InnoDB缓冲池里包含着数据的页面的个数。这个计数值既包括从没被修改过的“干净”页面，也包括其内容数据曾被修改过的“脏”页面。
- Innodb_buffer_pool_pages_dirty
  InnoDB缓冲池里内容数据曾被修改过的页面的个数。
- Innodb_buffer_pool_pages_flushed
  InnoDB缓冲池里内容数据曾经被写入硬盘的页面的个数。
- Innodb_buffer_pool_pages_free
  InnoDB缓冲池里可用页面的个数。
- Innodb_buffer_pool_pages_latched
  InnoDB缓冲池里因为正在被读、写或是因为其他一些原因而不能被转储并释放以便重复使用的页面的个数。
- Innodb_buffer_pool_pages_misc
  InnoDB缓冲池里为执行各种内部操作而分配的页面的个数。
- Innodb_buffer_pool_pages_total
  InnoDB缓冲池里的页面的总数。
- Innodb_buffer_pool_read_ahead_rdn
  InnoDB存储引擎曾执行过的随机预读操作的次数。这种读操作发生在InnoDB存储引擎需要在数据表里随机读取大段数据的时候。
- Innodb_buffer_pool_read_ahead_seq
  InnoDB存储引擎曾执行过的顺序预读操作的次数。这种读操作发生在InnoDB存储引擎需要在数据表里按顺序进行全表扫描的时候。
- Innodb_buffer_pool_read_requests
  InnoDB存储引擎发出逻辑读请求的次数。
- Innodb_buffer_pool_reads
  因为不能通过执行逻辑读操作的办法从InnoDB缓冲池获得所需数据而导致的单页面读操作（每次只读取一个页面）的个数。
- Innodb_buffer_pool_wait_free
  InnoDB存储引擎等待其数据缓冲池腾出可用页面（办法是把缓存在有关页面里的信息转储到磁盘）的次数。写操作通常是在后台进行的，但如果在需要读取或创建页面时没有页面可用，InnoDB存储引擎必须等待。
- Innodb_buffer_pool_write_requests

对InnoDB缓冲池进行写操作的请求的个数。

- Innodb_data_fsyncs
  已经完成的InnoDB数据“同步到磁盘”操作的个数。
- Innodb_data_pending_fsyncs
  正在等待执行的InnoDB数据“同步到磁盘”操作的个数。
- Innodb_data_pending_reads
  正在等待执行的InnoDB数据读操作的个数。
- Innodb_data_pending_writes
  正在等待执行的InnoDB数据写操作的个数。
- Innodb_data_read
  已经完成的InnoDB数据读操作所涉及的字节总数。
- Innodb_data_reads
  已经完成的InnoDB数据读操作的个数。
- Innodb_data_writes
  已经完成的InnoDB数据写操作的个数。
- Innodb_data_written
  已经完成的InnoDB数据写操作所涉及的字节总数。
- Innodb_dblwr_pages_written
  已被写到InnoDB双写缓冲区的页面的个数。
- Innodb_dblwr_writes
  已经对InnoDB双写缓冲区进行了多少次写操作。
- Innodb_log_waits
  InnoDB存储引擎等待其日志缓冲池腾出可用页面（办法是把缓存在有关页面里的信息转储入磁盘）的次数。
- Innodb_log_write_requests
  对InnoDB日志文件进行写操作的请求的个数。
- Innodb_log_writes
  已经完成的InnoDB日志文件写操作的个数。
- Innodb_os_log_fsyncs
  已经完成的InnoDB日志文件“同步到磁盘”操作的个数。
- Innodb_os_log_pending_fsyncs
  正在等待执行的InnoDB日志文件“同步到磁盘”操作的个数。
- Innodb_os_log_pending_writes
  正在等待执行的InnoDB日志文件写操作的个数。
- Innodb_os_log_written
  已经完成的InnoDB日志文件写操作涉及的字节总数。
- Innodb_page_size
  InnoDB存储引擎所使用的页面长度，这个值是在MySQL软件编译阶段设定的。我们可以利用这个变量把一个以页面为单位的统计结果转换为一个以字节为单位的长度值。默认值是16KB。

- Innodb_pages_created
  由InnoDB存储引擎创建的页面的总数。
- Innodb_pages_read
  已经完成的InnoDB读操作所涉及的页面总数。
- Innodb_pages_written
  已经完成的InnoDB写操作所涉及的页面总数。
- Innodb_row_lock_current_waits
  InnoDB存储引擎正在等待获得的数据行锁定的个数。这个变量是从MySQL 5.0.3版开始引入的。
- Innodb_row_lock_time
  InnoDB存储引擎在每次申请数据行级锁定时等待的总时间（以毫秒为单位）。这个变量是从MySQL 5.0.3版开始引入的。
- Innodb_row_lock_time_avg
  InnoDB存储引擎在每次申请数据行级锁定时等待的平均时间（以毫秒为单位）。这个变量是从MySQL 5.0.3版开始引入的。
- Innodb_row_lock_time_max
  InnoDB存储引擎在每次申请数据行级锁定时等待的最长时间（以毫秒为单位）。这个变量是从MySQL 5.0.3版开始引入的。
- Innodb_row_lock_waits
  InnoDB存储引擎经过Innodb_row_lock_waits等待之后才获得一个数据行锁定。这个变量是从MySQL 5.0.3版开始引入的。
- Innodb_rows_deleted
  从InnoDB数据表删除的数据行的个数。
- Innodb_rows_inserted
  被插入InnoDB数据表的数据行的个数。
- Innodb_rows_read
  从InnoDB数据表读出的数据行的个数。
- Innodb_rows_updated
  InnoDB数据表里被更新的数据行的个数。

D.3.2　与查询缓存有关的状态变量

下列变量可以提供关于查询缓存操作的信息。

- Qcache_free_blocks
  查询缓存里的可用内存块的个数。
- Qcache_free_memory
  查询缓存里的可用内存量。
- Qcache_hits
  有Qcache_hits个查询请求是通过这个缓存里的查询命令来满足的。
- Qcache_inserts

在查询缓存里注册过的查询命令的总数。

- `Qcache_lowmem_prunes`
  为了给新到的查询结果腾地方而被“踢”出查询缓存的老查询结果的个数。
- `Qcache_not_cached`
  无法被缓存或者因用户使用了`SQL_NO_CACHE`选项而没有被缓存的查询命令的个数。
- `Qcache_queries_in_cache`
  查询缓存里现在注册有多少条查询命令。
- `Qcache_total_blocks`
  查询缓存里总共有多少个内存块。

D.3.3　与SSL有关的状态变量

以下变量可以提供与 SSL 管理代码有关的信息。它们当中有许多只能用来了解当前连接的状态，如果当前连接没有使用 SSL 加密，它们将是空白的。如果 MySQL 服务器在编译时没有包括 SSL 支持机制，这些变量将不可用。

- `Ssl_accept_renegotiates`
  在服务器模式里开始重新协商（renegotiation）过程的次数。
- `Ssl_accepts`
  在服务器模式里开始SSL/TLS握手过程的次数。
- `Ssl_callback_cache_hits`
  在服务器模式里从外部会话缓存成功地检索到的会话的个数。
- `Ssl_cipher`
  当前连接所使用的SSL加密协议（若当前不存在生效的加密协议，则为空）。你可以利用这个变量来判断当前连接是否是加密的。
- `Ssl_cipher_list`
  当前有哪些SSL加密协议可供选用。
- `Ssl_client_connects`
  在客户模式里开始SSL/TLS握手过程的次数。
- `Ssl_connect_renegotiates`
  在客户（程序）模式里开始重新协商过程的次数。
- `Ssl_ctx_verify_depth`
  SSL上下文的验证深度。
- `Ssl_ctx_verify_mode`
  SSL上下文的验证模式。
- `Ssl_default_timeout`
  SSL会话默认的超时时间。
- `Ssl_finished_accepts`
  在服务器模式里成功地建立起来的SSL/TLS会话的个数。
- `Ssl_finished_connects`
  在客户模式里成功地建立起来的SSL/TLS会话的个数。

- `Ssl_session_cache_hits`
  在SSL会话缓存里成功地检索到的SSL会话的个数。
- `Ssl_session_cache_misses`
  没能在SSL会话缓存里成功地检索到的SSL会话的个数。
- `Ssl_session_cache_mode`
  MySQL服务器所使用的SSL机制的类型。
- `Ssl_session_cache_overflows`
  因SSL会话缓存已满而被切换出该缓存的会话的个数。
- `Ssl_session_cache_size`
  SSL会话缓存的容量，即它最多能够容纳多少个会话。
- `Ssl_session_cache_timeouts`
  因超时而被切换出SSL会话缓存的会话的个数。
- `Ssl_session_reused`
  当前会话是不是此前某个会话的再次使用。
- `Ssl_used_session_cache_entries`
  SSL会话缓存里现在容纳着多少个SSL会话。
- `Ssl_verify_depth`
  SSL验证深度。
- `Ssl_verify_mode`
  SSL验证模式。
- `Ssl_version`
  当前连接所使用的SSL协议版本。

D.4　用户定义变量

用户定义变量（简称“用户变量”）可以被赋值，还可以在后面的其他语句里引用。

用户定义变量的名字由“@”字符和紧随其后的一个标识符构成，需要遵守的规则和 MySQL 标识符差不多（请参阅 2.2 节）。值得注意的是用户变量名可以包含小数点（.）而无需用引号括起来，这是它们与标识符不一样的地方。用户变量的名字在 MySQL 5.0 之前的版本里是区分大小写的，在那以后的版本里是不区分大小写的。

用户变量可以在 SET 语句里用“=”或“:=”操作符来赋值，也可以在其他语句（如 SELECT 语句）里用“:=”操作符来赋值。我们可以在同一条语句里对多个变量进行赋值。如下所示：

```
mysql> SET @x = 0, @y = 2;
mysql> SET @color := 'red', @size := 'large';
mysql> SELECT @x, @y, @color, @size;
+------+------+--------+-------+
| @x   | @y   | @color | @size |
+------+------+--------+-------+
| 0    | 2    | red    | large |
+------+------+--------+-------+
mysql> SELECT @count := COUNT(*) FROM member;
```

```
+--------------------+
| @count := COUNT(*) |
+--------------------+
|                102 |
+--------------------+
```

用户变量可以被赋值为整数、小数、浮点数、字符串或 NULL 值，还可以通过任意形式的表达式赋值，而表达式里还允许出现其他变量。如果在访问某个用户变量之前没有对它明确赋值，它的值将是 NULL。

用户变量的值只在当前会话里有效，一旦与 MySQL 服务器的连接断开，那些值就将不复存在。

在返回多个数据行的 SELECT 语句里，对变量的赋值操作将依次使用每一个数据行来进行，查询结束时的变量值将是来自最后一个结果数据行的值。

对于字符串类型的用户变量，其字符集和排序方式与赋给它们的字符串相同：

```
mysql> SET @s = CONVERT('abc' USING latin2) COLLATE latin2_czech_cs;
mysql> SELECT CHARSET(@s), COLLATION(@s);
+-------------+-----------------+
| CHARSET(@s) | COLLATION(@s)   |
+-------------+-----------------+
| latin2      | latin2_czech_cs |
+-------------+-----------------+
```

附录 E

SQL 语法指南

本附录描述 MySQL 提供的 SQL 语句的语法，包括下面 3 个部分。

- 不同于复合语句的 SQL 语句的语法。
- 用于复合语句的 SQL 语句，用 BEGIN 和 END 写成，可用于编写存储在服务器端的程序（函数、过程、触发器和事件）。
- 在 SQL 代码里编写注释的语法。注释用于编写不会被 MySQL 服务器执行的描述性文字或隐藏 MySQL 特有的关键字（这些关键字会被 MySQL 执行，被其他数据库服务器忽略）。

在介绍 SQL 语法时，本附录使用了下面一些记号。

- 可选信息将放在一对方括号（[]）里。
- 垂直线字符（|）用来分隔参数清单里的多选一可选项。若参数清单出现在一对方括号里，则表示可以选取；如果参数清单出现在一对花括号（{}）里，则表示必须选取。
- 省略号（...）表示该省略号前面的那个项目可以重复多次地出现。
- *n* 代表一个整数。
- '*str*'代表一个字符串值。单引号串值'*file_name*'或'*pattern*'表示更特定类型的值，比如一个文件名或者一个模式。

如果未做特别说明，本附录列出的 SQL 语句至少从 MySQL 5.0.0 版本起就已经出现在 MySQL 里了。在那以后新增加的或是含义发生了变化的语句将在有关内容里加以说明。

有些语句已经过时，将被摈弃或者（以我之愚见）只有极其有限的用途，所以我没有把它们收录在这里：

```
BACKUP TABLE
LOAD DATA FROM MASTER
LOAD TABLE tbl_name FROM MASTER
RESTORE TABLE
SHOW AUTHORS
SHOW CONTRIBUTORS
```

本附录没有收录与插件、用户定义函数（User-Defined Function，简称 UDF）、XA 事务、MySQL Cluster 专用的语句或语句子句。

我在这里列出一些通用的同义词，在后面的有关内容里就不再列出它们了：

如果需要指定一个字符集，可以使用以下格式中的任何一种：

```
CHARACTER SET charset
CHARSET = charset
```

```
CHARSET charset
```

这些含义相同的语法形式既可以用在数据表和数据列的定义里，也可以用在 CREATE DATABESE 和 ALTER DATABASE 语句里。

从 MySQL 5.0.2 版开始，SCHEMA 和 SCHEMAS 分别是 DATABASE 和 DATABASES 的同义词，在允许使用后两个关键字的任何语句里都可以随意替换。比如说，如果你想创建一个数据库，使用 CREATE DATABASE 语句和使用 CREATE SCHEMA 语句将没有任何区别。

E.1 SQL 语句

本节将对 MySQL 支持使用的各 SQL 语句（复合语句除外，参见 E.2 节）的语法和含义进行描述。给定一条 SQL 语句，如果你没有足够的权限去执行它，这条语句的执行就会失败。比如说，如果你没有访问 *db_name* 数据库的权限，你发出的 USE *db_name* 语句就不会成功执行。

- ALTER DATABASE

```
ALTER DATABASE [db_name] db_attr ...

ALTER DATABASE db_name UPGRADE DATA DIRECTORY NAME
```

这条语句用于改变数据库特性或者修改数据库目录名的编码，它要求具备数据库的 ALTER 权限才能执行成功。

对于第一种语法，允许的 *db_attr* 特性值和在 CREATE DATABASE 项中列出的一样。如果省略数据库名，这条语句用于默认数据库。如果没有默认数据库，则会发生错误。

从旧版本更新至 MySQL 5.1 或更高版本时，需要使用 UPGRADE DATA DIRECTORY NAME 语法，如果 MySQL 的文件系统编码方式需要的话，它会对数据库目录名称重新编码（如果名称包含特定字符）。这个语法是从 MySQL 5.1.23 开始引入的。

- ALTER EVENT

```
ALTER
  [DEFINER = definer_name]
  EVENT event_name
  [ON SCHEDULE schedule]
  [ON COMPLETION [NOT] PRESERVE]
  [RENAME TO new_event_name]
  [ENABLE | DISABLE [ON SLAVE]]
  [COMMENT 'str']
  [DO event_stmr]
```

这条语句用于更改现有事件，使其具备给定定义。RENAME TO 子句对事件进行重命名。在后面的 CREATE EVENT 项中，将描述其他子句。必须具备事件所属数据库的 EVENT 权限才能使用这条语法。（在 MySQL 5.1.12 之前，你必须具备 SUPER 权限或者是事件的定义者。）

- ALTER FUNCTION，ALTER PROCEDURE

```
ALTER {FUNCTION | PROCEDURE} routine_name [characteristic] ...

characteristic:
  [NOT] DETERMINISTIC
  | LANGUAGE SQL
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT 'str'
```

这些语句负责更改存储例程的特征，后面介绍 CREATE FUNCTION 和 CREATE PROCEDURE 时将介绍这些特征。

从 MySQL 5.0.3 开始，这些语句要求具备给定例程的 ALTER ROUTINE 权限。

- ALTER SERVER

```
ALTER SERVER server_name OPTIONS (option [, option] ...)
```

这条语句用于修改 FEDERATED 表服务器 *server_name* 的定义，更新 mysql.servers 表的相应数据行。省略的选项仍为以前的值。必须具备 SUPER 权限才能使用这条语句。

参见后面介绍的 CREATE SERVER，可了解 OPTIONS 子句的允许值。这个语句是从 MySQL 5.1.15 开始引入的。

- ALTER TABLE

```
ALTER [IGNORE] TABLE tbl_name action [, action] ...
```

ALTER TABLE 用于重新命名一个数据表或者更改它的结构。使用这条命令时，需要给出数据表名 tbl_name 以及一个或者多个将对该数据表进行的操作。如果某个操作会使新数据表里的唯一化索引出现重复的键值，就需要选用 IGNORE 关键字。如果没有这个 IGNORE 关键字，ALTER TABLE 语句的效果将被撤销；如果有这个关键字，会导致唯一化索引出现重复键值的那些数据行被删除掉。

除重命名数据表之外，ALTER TABLE 语句将先根据原始数据表进行修改，创建一个新数据表。如果执行出错，新数据表将被丢弃而原始数据表则保持不变。如果全部操作执行成功，新数据表就将取代原始数据表（原始数据表将被丢弃）。在这一过程中，其他客户（程序）依然可以从原始数据表里读取数据，但试图修改它的客户（程序）将被阻塞，直到 ALTER TABLE 语句执行完毕，修改操作将实施在新数据表上。

action 值用来给出具体的修改操作，这些操作将被依次执行。有些修改操作不能与其他操作互相组合，我们将在各操作的具体描述里注明。

对于包含 *index_type* 或 *index_option* 子句的索引定义操作，有几种存储引擎可用于指定索引算法或其他索引定义限定符。在关于 CREATE INDEX 语句的条目里，对不同 MySQL 版本所允许使用的索引值有详细的说明。与索引创建工作有关的其他信息请参阅 2.6.4 节。

action 值可以是下面的任何一种：

- *table_option*

  用来给出在 CREATE TABLE 语句的 *table_option* 部分允许使用的数据表选项，例如：

  ```
  ALTER TABLE score ENGINE = MyISAM CHECKSUM = 1;
  ALTER TABLE sayings CHARACTER SET utf8;
  ```

  关于不同的 MySQL 版本和不同的存储引擎都允许使用哪些数据表选项，请见 CREATE TABLE 语句条目中的描述。如果试图通过 ALTER TABLE 语句让某个数据表改用另一种存储引擎、但该存储引擎不可用，该语句的最终效果将取决于 NO_ENGINE_SUBSTITUTION SQL 模式的设置值。从 MySQL 5.0.23/5.1.11 版本开始，让数据表改用 MERGE 或 BLACKHOLE 存储引擎是不允许的，因为那样有可能导致数据丢失。

  请注意，[DEFAULT] CHARACTER SET 数据表选项可以改变默认的数据表字符集，但不能把现有的数据列转换为新字符集。如果想执行后一种操作，必须使用一个 CONVERT TO CHARACTER SET 操作。

  ALTER TABLE 语句忽略 DATA DIRECTORY 和 INDEX DIRECTORY 数据表选项。

- ❑ ADD [COLUMN] *col_name col_declaration* [FIRST | AFTER *col_name*]
  给数据表增加一列。*col_declaration* 是新增数据列的定义，*col_name* 是其名称，它与 CREATE TABLE 语句中的数据列定义格式完全相同。如果给出了 FIRST 关键字，新增数据列将成为该数据表的第一个数据列；如果给出了 AFTER *col_name*，新增数据列将被安排在数据列 *col_name* 的后面；如果没有替新增数据列安排位置，它将成为该数据表的最后一列。

```
ALTER TABLE t ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY;
ALTER TABLE t ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY FIRST;
ALTER TABLE t ADD id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY
  AFTER suffix;
```

- ❑ ADD [COLUMN](*create_definition*,...)
  给数据表增加数据列或索引。每个 *create_definition* 都是一个数据列或索引定义，格式则与 CREATE TABLE 语句中的定义格式完全相同。

- ❑ ADD [CONSTRAINT [*name*]] FOREIGN KEY *[fk_name]*
  (*index_columns*) *reference_definition*

  为给定的数据表增加一个外键定义。这种变更操作只适用于 InnoDB 数据表。新外键由 *index_columns* 列表里列出的数据列构成，该列表由给定数据表里的一个或多个以逗号分隔的数据列构成。如果给出了 CONSTRAINT 部分，其中列出的任何名字都将被忽略。*fk_name* 是新外键的 ID，除非 InnoDB 存储引擎自动为新外键创建了一个索引——此时 *fk_name* 将成为该索引的名字，否则 *fk_name* 将被忽略。*reference_definition* 部分用来定义新外键与父数据表的关系，相关语法见 CREATE TABLE 语句条目里的描述。

```
ALTER TABLE child
  ADD FOREIGN KEY (par_id) REFERENCES parent (par_id) ON DELETE CASCADE;
```

  ADD FOREIGN KEY 和 DROP FOREIGN KEY 操作不能出现在同一条 ALTER TABLE 语句里。

- ❑ ADD FULLTEXT [INDEX | KEY] [*index_name*]
  (*index_columns*) [*index_option*] ...
  给 MyISAM 数据表增加一个 FULLTEXT 索引。*index_columns* 是一个或者多个彼此以逗号分隔的非二进制字符串数据列名字，新增加的这个 FULLTEXT 索引就将建立在这些数据列上。ADD FULLETXT 语法最早出现于 MySQL 3.23.23 版本。

```
ALTER TABLE poetry ADD FULLTEXT (author,title,stanza);
```

- ❑ ADD {INDEX | KEY} [*index_name*] [*index_type*]
  (*index_columns*) [*index_option*] ...
  给数据表增加一个索引。*index_columns* 是一个或者多个彼此以逗号分隔的数据列名字，新增加的这个索引就建立在这些数据列上。如果你没有给出索引名字，MySQL 就将自动使用第一个被索引数据列的名字来作为这个索引的名字。

- ❑ ADD [CONSTRAINT [*name*]] PRIMARY KEY [*index_type*]
  (*index_columns*) [*index_option*] ..
  在给定数据列上建立一个主键。这个主键的名字是 PRIMARY。*index_columns* 参数的用途和用法与它在 ADD INDEX 子句里的情况相同。每列必须定义为 NOT NULL。如果主键已经存在，这个操作将报告出错。

```
ALTER TABLE president ADD PRIMARY KEY (last_name, first_name);
```

- ADD SPATIAL [INDEX | KEY] [*index_name*]
  (*index_columns*) [*index_option*]...

  为给定的 MyISAM 数据表增加一个 SPATIAL 索引。新索引由 *index_columns* 列表里列出的数据列构成，该列表由给定数据表里的一个或多个以逗号分隔的空间数据列构成，这些数据列中的每一个都必须被定义为 NOT NULL。*index_name* 部分的定义和 ADD INDEX 操作中的情况一样。

```
ALTER TABLE coordinates ADD SPATIAL (x,y);
```

- ADD [CONSTRAINT [name]] UNIQUE [INDEX | KEY]
  [*index_name*] [*index_type*]
  (*index_columns*) [*index_option*]...

  为 *tbl_name* 数据表增加一个 UNIQUE 索引。*index_name* 和 *index_columns* 部分的定义与 ADD INDEX 操作中的情况一样。

```
ALTER TABLE absence ADD UNIQUE id_date (student_id, date);
```

- ALTER [COLUMN] *col_name* { SET DEFAULT *value* | DROP DEFAULT }

  改变指定数据列的默认值。这个子句既可以用来设定一个新的默认值，也可以丢弃当前的默认值。在后一种情况里，新默认值将按照 3.2.3 节中的描述进行设定。

```
ALTER TABLE grade_event ALTER category SET DEFAULT 'Q';
ALTER TABLE grade_event ALTER category DROP DEFAULT;
```

- CHANGE [COLUMN] *old col_name new_col_name col_declaration*
  [FIRST | AFTER *col_name*]

  改变指定数据列的名称和定义。*old_col_name* 和 *new_col_name* 分别是该数据列现在的和新的名字；*col_declaration* 则是该数据列的新定义，其格式与 CREATE TABLE 语句所使用的格式相同，包括所有的列特性，如 NULL 和 NOT NULL。注意如果，想改变定义但不想改变它的名字，就必须把它现在的名字写两遍。关键字 FIRST 或 AFTER 的用途与 ADD COLUMN 子句相同。

```
ALTER TABLE student CHANGE name name VARCHAR(40);
ALTER TABLE student CHANGE name student_name CHAR(30) NOT NULL;
```

- CONVERT TO CHARACTER SET *charset* [COLLATE *collation*]

  把给定数据表的默认字符集和所有的非二进制字符数据列转换为给定的新字符集。binary 将把各数据列转换为相应的二进制字符串数据类型。DEFAULT 将把数据表转换为使用其所在数据库的字符集。COLLATE 子句同样可以用来指定一种排序方式。如果省略了 COLLATE 子句，将使用新字符集的默认排序方式。

- DISABLE KEYS

  对于一个 MyISAM 数据表，它的非 UNIQUE 索引通常会在该数据表发生变化时及时更新，但这个操作将禁止这种更新行为。ENABLE KEYS 操作可以用来重新启用索引更新功能。

```
ALTER TABLE score DISABLE KEYS;
```

- DISCARD TABLESPACE

  这个操作适用于有专用数据表空间的 InnoDB 数据表。对于一个这样的数据表，本操作将删除用于存储数据表内容的 *tbl_name*.idb 文件。这个操作不能与其他操作搭配使用。

- DROP [COLUMN] *col_name* [RESTRICT | CASCADE]

  从数据表里删除指定的数据列。同时，如果该数据列还是某个索引的组成部分，它也将从该索

引中被剔除掉。如果构成某个索引的所有数据列全被剔除了，该索引也将被删除。

```
ALTER TABLE president DROP suffix;
```

如果给出，RESTRICT 和 CASCADE 关键字会被解析但会被忽略，因而没有任何实际效果。

- DROP FOREIGN KEY *fk_name*
  丢弃具有给定名字的外键定义。ADD FOREIGN KEY 和 DROP FOREIGN KEY 操作不能出现在同一条 ALTER TABLE 语句里。
- DROP {INDEX | KEY } *index_name*
  从数据表删除给定的索引。

```
ALTER TABLE member DROP INDEX name;
```

- DROP PRIMARY KEY
  从数据表删除主键。如果该数据表根本没有主键，这个操作将报告出错。

```
ALTER TABLE president DROP PRIMARY KEY;
```

- ENABLE KEYS
  对于 MyISAM 数据表，重新启用被 DISABLE KEYS 子句禁用的非唯一索引自动更新机制。

```
ALTER TABLE score ENABLE KEYS;
```

- IMPORT TABLESPACE
  这个操作适用于有专用数据表空间的 InnoDB 数据表。对于这样的数据表，本操作将把该数据表所在的数据库目录里的 *tbl_name*.idb 文件与该数据表关联起来（本操作的前提条件是该数据表原先的.ibd 文件已经被 DISCARD TABLESPACE 操作删掉了）。这个操作不能与其他操作搭配使用。
- MODIFY [COLUMN]*col_name col_declaration* [FIRST | AFTER *col_name*]
  改变数据列的定义。*col_name* 是待修改的列名。数据列定义 *col_declaration* 的格式与 CREATE TABLE 语句所使用的格式相同，包括所有的列特性，如 NULL、NOT NULL 和 DEFAULT。FIRST 和 AFTER 的效果与 ADD COLUMN 的情况相同。

```
ALTER TABLE student MODIFY name VARCHAR(40) DEFAULT '' NOT NULL;
```

- ORDER BY *col_list*
  col_list 是一个或者多个彼此以逗号分隔的数据列名字，数据表里的数据行将根据它们排序。默认的排序顺序是升序。你可以在各数据列名字的后面加上关键字 ASC 或 DESC 来明确地指定按升序或降序进行排序。对数据表进行排序能够提高按同样顺序执行的数据行检索速度。不过，如果数据表在进行完 ORDER BY 操作之后又发生了修改，就会打乱刚才排好的顺序，所以这个操作只对那些今后不再会发生修改的数据表才有用。

```
ALTER TABLE score ORDER BY event_id, student_id;
```

- RENAME [TO | AS] *new_tbl_name*
  把 *tbl_name* 数据表重新命名为 new_*tbl_name*。如果重新命名的是一个 InnoDB 数据表，其他的数据表又依赖于该 InnoDB 数据表的外键关系，InnoDB 存储引擎将调整那些依赖关系指向重新命名后的数据表。

```
ALTER TABLE president RENAME TO prez;
```

从 MySQL 5.1 版开始，ALTER TABLE 语句还可以用来调整分区设置。关于 CREATE TABLE 语句的条目对以下操作描述中的 *partition_scheme* 和 *partition_definition* 术语的含义作出了定义。

- *partition_scheme*
  根据给定的分区描述对数据表进行分区。如果数据表此前没有分区，它将变成有分区的；如果此前已有分区，新分区将取代老分区。
- ADD PARTITION [*partition_definition*]
  给一个已经有分区的数据表增加一个新的分区。
- COALESCE PARTITION *n*
  把一个分区数据表的分区个数缩减为 *n* 个。这个操作只适用于 HASH 和 KEY 分区，被删除的分区里的数据将合并到留下来的分区里。如果想删除 LIST 或 RANGE 分区，应该使用 DROP PARTITION 操作。
- [DROP | REBUILD] PARTITION *partition_name* [, *partition_name*]...
  对给定的分区进行指定的操作。DROP 操作只适用于 LIST 或 RANGE 分区，被丢弃的分区里的数据将丢失。如果只是想减少 HASH 或 KEY 分区的数量，应该使用 COALESCE PARTITION 操作。
- REMOVE PARTITIONING
  删除所有的分区，你将得到一个未分区的数据表。这个选项是从 MySQL 5.1.8 版开始引入的。(在 MySQL 5.1.8 版之前，可以通过 ALTER TABLE 语句的 ENGINE 数据表选项来删除某个分区数据表的所有分区。)
- REORGANIZE PARTITION *partition_name* [, *partition_name*]...
  INTO (*partition_definition [, partition_ definition]...*)
  使用新的分区定义对给定的分区重新进行分区。
  以下分区选项不允许同时用在同一条 ALTER TABLE 语句里：*partition_scheme*、ADD PARTITON、COALESCE PARTITON、DROP PARTITON、REORGANIZE PARTITON。

- ALTER VIEW

```
ALTER
  [ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]
  [DEFINER = definer_name]
  [SQL SECURITY = {DEFINER | INVOKER}]
  VIEW view_name [(col_list)] AS select_stmt
  [WITH [CASCADED | LOCAL] CHECK OPTION]
```

根据给定的定义修改现有的视图。各个子句的含义与 CREATE VIEW 语句条目所描述的相同。

执行 ALTER VIEW 语句需要具备 CREATE VIEW 权限、相关视图元素的 DROP 权限，以及在定义视图的 SELECT 语句里用到的每一个数据列上的必要权限。从 MySQL 5.0.25/5.1.23 版开始，ALTER VIEW 语句只允许视图的定义者或是具备 SUPER 权限的用户使用。

ALTER VIEW 语句是从 MySQL 5.0.1 版开始引入的。DEFINER 和 SQL SECURITY 子句是从 5.0.16 版开始引入的。

- ANALYZE TABLE

```
ANALYZE
  [LOCAL | NO_WRITE_TO_BINLOG]
  {TABLE | TABLES} tbl_name [, tbl_name] ...
```

让 MySQL 对数据表进行分析，把数据表各索引的键值分布情况统计并保存起来。它适用于 MyISAM 和 BDB 数据表，要求你必须拥有指定数据库上的 SELECT 和 INSERT 权限。在分析工作完成之后，SHOW INDEX 输出报告里的 Cardinality 列将给出索引里有多少彼此不同的值。利用这条语句得到的分析结果，优化器就能在今后的查询里更快地完成某些特定的联结操作。

数据表的分析操作需要用到一个读操作锁，在分析期间，对数据表的写操作将被阻塞。如果你已经对某个数据表进行过分析且在分析工作完成后还没有修改过，再次发出一条 ANALYZE TABLE 命令将不会再次对之分析。

ANALYZE TABLE 语句产生的输出报告的格式与 CHECK TABLE 语句相同，请参阅有关条目。

如果启用了二进制日志记录，MySQL 会对二进制日志编写 ANALYZE TABLE 语句，除非给定了 LOCAL 或 NO_WRITE_BINLOG 选项。

- BEGIN

```
BEGIN [ WORK ]
```

这个语句是 START TRANSACTION 的同义词，参见这条语句的介绍。

BEGIN 也可以和 END 用在存储程序里，用于创建复合语句，参见 E.2 节。

- CACHE INDEX

```
CACHE INDEX
  tbl_name [[INDEX | KEY] (index_name [, index_name] ...)]
  [, tbl_name [[INDEX | KEY] (index_name [, index_name] ...)]] ...
  IN cache_name
```

把一个或多个 MyISAM 数据表与给定的键缓存关联起来，该键缓存必须已经存在。你必须在这条语句所涉及的每一个数据表上具备 INDEX 权限。默认的键缓存区的名字是 default。数据表的索引可以稍后用 LOAD INDEX 语句加载到相应的缓存里。就目前而言，虽然 CACHE INDEX 语句的语法允许只为特定的索引分配键缓存，但实际情况却是该语句将把各数据表的所有索引全都关联到指定的缓存去。只为特定的索引单独分配一个键缓存的关联操作还有待实现。

下面的语句将为 member 数据表的索引分配一个名为 member_cache 的键缓存：

```
CACHE INDEX member IN member_cache;
```

CACHE INDEX 语句的输出内容的格式与 CHECK TABLE 语句条目里描述的一样。

有关 MyISAM 键缓存管理的更多信息，请参阅 12.7.2 节。

- CALL

```
CALL routine_name([proc_param [, proc_param] ...])
CALL routine_name[()]
```

调用有给定名字的存储过程。可选的参数列表由一个或多个以逗号分隔的参数值构成。只要参数列表里有 OUT 或 INOUT 参数，被调用的存储过程就可以通过它们返回值。

当某个存储过程返回时，我们可以通过调用 ROW_COUNT() 函数获得它最近执行过的修改数据行的语句所影响的行数。在 C 语言程序里，我们可以通过调用 mysql_affected_rows() 函数来获得这个值。

从 MySQL 5.0.30/5.1.13 版开始，如果某个存储过程不带任何参数，该过程名字后面的括号允许省略。

- CHANGE MASTER

```
CHANGE MASTER TO option [, option] ...
```

在主从复制机制中的从服务器上改变其配置参数，比如将要连接到哪一个主服务器、如何与之建立连接、使用哪些日志，等等。参数保存在从服务器的 master.info 和 relay_log.info 文件中，用于随后的从服务器重启。

每个 *option* 指定一个参数，参数的定义格式为 *param* = *value*。它可以用来设置以下参数：

- MASTER_CONNECT_RETRY = *n*
  试图连接主服务器的各次尝试之间的等待时间间隔，以秒为计算单位。
- MASTER_HOST = '*host_name*'
  主服务器所在的主机名。
- MASTER_LOG_FILE = '*file_name*'
  主服务器的某个二进制日志的文件名，从服务器将使用这个文件去建立复制机制。
- MASTER_LOG_POS = *n*
  主服务器日志文件中的某个位置，从服务器将从这个位置开始或者（在中断后）继续去建立复制机制。
- MASTER_PASSWORD = '*pass_val*'
  连接主服务器时使用的口令。
- MASTER_PORT = *n*
  连接主服务器时使用的 TCP/IP 端口号。
- MASTER_SSL = {0 | 1}

  ```
  MASTER_SSL_CA = 'file_name'
  MASTER_SSL_CAPATH = 'dir_name'
  MASTER_SSL_CERT = 'file_name'
  MASTER_SSL_CIPHER = 'str'
  MASTER_SSL_KEY = 'file_name'
  MASTER_SSL_VERIFY_SERVER_CERT = {0 | 1}
  ```

  这些选项指定的是与主服务器建立 SSL 连接时的参数，它们与 F.1.2 节中第 1 小节描述的 *ssl-xxx* 选项具有相同含义。它们保存在 master info 文件中，但不会产生影响，除非从服务器启用了 SSL 支持。
- MASTER_USER = '*user_name*'
  连接主服务器时使用的用户名。
- RELAY_LOG_FILE = '*file_name*'
  从服务器的中继日志（relay log）的文件名。
- RELAY_LOG_POS = *n*
  从服务器中继日志文件里的当前位置。

没有在语句里指定的参数将维持当前值，但有一种情况除外：更改 MASTER_HOST 或 MASTER_ PORT 通常表明更改了从服务器，所以如果指定这些选项中的某一个，MASTER_LOG_FILE 和 MASTER_LOG_POS 值就会被设置成主服务器的第一个二进制日志文件的开头部分。

不能在同一语句中混合使用 MASTER_LOG_FILE、MASTER_LOG_POS 选项与 RELAY_LOG_FILE、RELAY_LOG 选项。

CHANGE MASTER 语句删除所有现存的中继日志文件，开始一个新的文件，除非指定了 RELAY_LOG_FILE 或 RELAY_LOG_POS 选项。

- CHECK TABLE

```
CHECK {TABLE | TABLES} tbl_name [, tbl_name] ... [option] ...
```

检查数据表有无错误。这条命令适用于 MySQL 5.0.16 之后的 MyISAM 数据表、InnoDB 表和 ARCHIVE 表，以及 MySQL 5.1.19 以后的 CSV 表。从 5.0.2 起，CHECK TABLE 可以检查视图定义的问题，如引用了已不存在的表，它要求你必须拥有有关数据表或视图的 SELECT 权限。

对于 MyISAM 表，CHECK TABLE 也更新索引统计信息。对于 InnoDB 表，如果服务器发现问题，会在错误日志里编写一条消息后中止运行，防止发生进一步的错误。

每个 options 值都可以是以下选项之一。除非另有说明，这些选项都应用于 MyISAM 表，而 InnoDB 表和视图会忽略它们，也有可能被其他存储引擎使用。

- ❑ CHANGED 只对上次检查后又发生过修改或者没有被正常关闭的数据表进行检查。
- ❑ EXTEND 进行扩展检查，以确保数据表完全没有错误。这是最全面的检查，但也是执行速度最慢的。比如说，它将检查每个索引中的每个键是否都指向一个数据行。
- ❑ FAST 只检查没有正常关闭的数据表。
- ❑ MEDIUM 检查索引、扫描数据行以检查是否有错误、进行校验和验证。如果没有在 options 部分给出任何选项，就将以此为默认值。
- ❑ QUICK 不扫描数据行，只检查索引。
- ❑ FOR UPGRADE 选项用来确定被检查的数据表是否与你当前使用的 MySQL 版本相兼容，所以在升级软件后这个选项很有用。只要发现有一处不兼容，MySQL 服务器就会进行一次全面检查。如果全面检查没有成功，应该尝试修复那个数据表。除非有不兼容之处并且全面检查未能成功，否则 MySQL 服务器将根据当前的 MySQL 版本对数据表的.frm 文件升级。这个选项的适用范围并不仅限于 MyISAM 数据表。它是从 MySQL 5.0.19/5.1.7 版开始引入的。

在没有使用 FOR UPGRADE 选项的情况下对数据表进行检查时，如果在检查某个 MyISAM 数据表时没有给出 QUICK、MEDIUM 或 EXTENDED 选项中的任何一个，CHECK TABLE 语句将默认使用 MEDIUM 选项去检查那些数据行长度可变的数据表。如果数据行长度是固定的，并且你给出了 CHANGED 或 FAST 选项，CHECK TABLE 语句将默认使用 QUICK 选项进行检查，否则，默认使用 MEDIUM 选项。

CHECK TABLE 语句会返回一些关于操作结果的信息，如下所示：

```
mysql> CHECK TABLE t;
+--------+-------+----------+----------+
| Table  | Op    | Msg_type | Msg_text |
+--------+-------+----------+----------+
| test.t | check | status   | OK       |
+--------+-------+----------+----------+
```

ANALYZE TABLE、CACHE INDEX、LOAD INDEX INTO CACHE、OPTIMIZE TABLE 和 REPAIR TABLE 等语句也会返回一些上述格式的信息。这些返回信息中的 Table 列告诉我们这次操作是在哪个数据表上进行的。Op 列告诉我们这条语句执行的是何种类型的操作。Msg_typt 和 Msg_text 列则是关于这次操作结果的信息；如果这两个输出列里的值表明数据表状态不佳或是尚未升级，你应该对之进行修复。

- CHECKSUM TABLE

```
CHECKSUM {TABLE | TABLES} tbl_name [, tbl_name] ...
 [QUICK | EXTENDED]
```

计算并报告某给定数据表的校验和。

```
mysql> CHECKSUM TABLE president;
+------------------+------------+
| Table            | Checksum   |
+------------------+------------+
| sampdb.president | 3032762697 |
+------------------+------------+
```

如果给定的数据表不存在，Checksum 值将是 NULL，并且（从 MySQL 5.0.3 版开始）会生成一条

警告消息。

在默认的情况下，只要存储引擎支持，这条语句所返回的将是实时校验和。(实时校验和会随着数据表的每次修改而更新。)对于MyISAM数据表,你可以通过在相应的CREATE TABLE或ALTER TABLE语句里加上 CHECKSUM = 1 选项来启用实时校验和更新。

如果你在 CHECKSUM TABLE 语句里使用了 QIUCK 选项，该语句将只在给定数据表确实有一个实时校验和的情况下才返回，其他情况都将返回 NULL。如果使用的是 EXTENDED 选项，MySQL 将读取整个数据表来计算其校验和并返回。这个操作会随着数据表体积的增加而变得越来越慢。

- COMMIT

```
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
```

提交当前事务里的各条语句对数据表的修改，把那些修改永久性地记录到数据库里。COMMIT 语句只适用于支持事务处理的存储引擎。(对于非事务型存储引擎，语句在执行完毕后会立刻被提交。)

可选关键字 WORK 目前没有任何效果。CHAIN 和 RELEASE 子句对 MySQL 服务器在事务结束时的处理有影响。如果使用了 AND CHAIN 子句，MySQL 服务器将在当前事务结束后以相同的隔离级别开始执行另一个交易。如果使用了 RELEASE 子句，MySQL 服务器将在当前事务结束后断开当前连接。给 CHAIN 或 RELEASE 子句加上 NO 将分别导致 MySQL 服务器不立刻执行一个新事务或是不断开当前连接。在没有这些子句的情况下，COMMIT 语句的行为将取决于系统变量 completion_type 的设置值。在默认的情况下，MySQL 服务器不应用 CHAIN 和 RELEASE。

如果没在事先通过 START TRANSACTION 语句或是通过把 autocommit 变量设置为 0 的办法禁用自动提交功能，COMMIT 语句将没有任何效果。

有些语句无法成为事务的组成部分，它们会隐式结束交易，就像执行了 COMMIT 语句那样。一般来说，那些用来创建、变更或删除数据库或数据库对象的 DDL（Data Definition Language，数据定义语言）语句以及与锁定机制有关的语句都有这样的效果。比如说，如果你在某个事务正在进行时发出了下列语句中的任何一个，MySQL 服务器就会在执行该语句之前先行提交当前事务：

```
ALTER TABLE
CREATE INDEX
DROP DATABASE
DROP INDEX
DROP TABLE
LOCK TABLES
RENAME TABLE
SET autocommit = 1 (if not already set to 1)
TRUNCATE TABLE
UNLOCK TABLES (if tables currently are locked)
```

从《MySQL 参考手册》可以查到在你当前使用的 MySQL 版本里都有哪些语句会导致 MySQL 服务器隐含地提交当前事务。

WORK、CHAIN 和 RELEASE 子句是从 MySQL 5.0.3 版开始引入的。

- CREATE DATABASE

```
CREATE DATABASE [IF NOT EXISTS] db_name [db_attr] ...

db_attr:
    [DEFAULT] CHARACTER SET [=] charset
  | [DEFAULT] COLLATE [=] collation
```

以给定名称创建一个数据库。如果你没有创建该数据库的 CREATE 权限，这条语句将执行失败。

一般说来，如果你打算创建的数据库已经存在，这条语句通常会执行失败并报告出错；但如果你给出了 IF NOT EXISTS 子句，数据库将不会被创建，这条语句也不会报告出错。

可选的 CHARACTER SET 和 COLLATE 属性可在数据库名之后给出，以指定一个默认的字符集和排列方式。这些属性用于没有显式指定字符集或排列方式的数据表。*chartset* 可以是字符集名；也可以是 DEFAULT，表示使用当前的服务器字符集。*collation* 可以是一个排列方式名称，或者为 DEFAULT，表示使用当前服务器的排列方式。

如果没有给出上述任何属性，新数据库将使用服务器级字符集和排序方式。如果只给出了 CHARACTER SET 属性但没有给出 COLLATE 属性，新建数据库将使用给定字符集的默认排序方式。如果只给出了 COLLATE 属性但没有给出 CHARACTER SET 属性，MySQL 服务器会根据你给定的排序方式确定一种字符集。如果打算同时给出 CHARACTER SET 和 COLLATE 属性，你给出的排序方式和字符集就必须是相兼容的。

MySQL 把数据库的属性保存在数据库子目录中的 db.opt 文件里。

- CREATE EVENT

```
CREATE
  [DEFINER = definer_name]
  EVENT [IF NOT EXISTS] event_name
  ON SCHEDULE schedule
  [ON COMPLETION [NOT] PRESERVE]
  [ENABLE | DISABLE | DISABLE ON SLAVE]
  [COMMENT 'str']
  [DO event_stmt]

schedule:
    AT datetime
  | EVERY expr interval [STARTS datetime] [ENDS datetime]
```

为事件调度程序创建一个名为 *event_name* 的新事件，必须在新事件所属的数据库上拥有 EVENT 权限才能创建它。在默认的情况下，新事件将被创建在默认数据库里。如果想在特定的数据库里创建一个新事件，必须以 *db_name.event_name* 的格式给出它的名字。

DEFINER子句的作用是在执行某个事件时确定其信息安全上下文（即用来核查其访问权限的账户），详细情况请参阅 4.5 节。默认的情况是使用当初执行CREATE EVENT语句的那个用户账户来执行该事件。

ON SCHEDULE 子句的作用是为新事件安排执行时间（假设事件调度程序正在运行中）。在这个子句各种各样的格式中，*datetime* 代表一个日期/时间值。CURRENT_TIMESTAMP()函数（或它的同义词）可以用来代表当前日期和时间。在 *datetime* 表达式里可以使用 INTERVAL *expr interval* 语法来加上或减去一个时间间隔。该语法在 C.2.5 节中的 DATE_ADD()函数条目里有详细的描述。这里的 interval 值不允许使用任何与微秒有关的限定符。

在 ON SCHEDULE 子句里，AT 形式的设置项将使得该事件在指定时刻执行且仅执行一次。EVERY 形式的设置项将使得该事件定期重复执行，重复间隔由一个数值和一个用来表明如何解释该数值的 *interval* 限定符构成（例如：5 HOUR 或'1:30' MINUTE_SECOND)。在默认的情况下，新事件的首次执行发生在它刚被创建时，然后再按照设定的时间间隔重复执行。STARTS 子句可以用来设定事件的首次执行时间。如果给出了 ENDS 子句，事件在该子句所设定的时间后将不再执行。ON SCHEDULE 子句只与时间有关，其中既不应该出现数据表引用，也不应该出现指向存储函数或用户定义函数的引用。

DO 子句用来给出事件发生时将要执行的语句。它应该只包含一条 SQL 语句。如果需要使用多条

语句，必须用关键字 BEGIN 和 END 把它们括起来以构成一个复合语句。（请参阅 E.2 节。）

在默认的情况下，事件在它最后一次执行完毕之后将被丢弃。ON COMPLETION NOT PRESERVE 子句明确地表明了这种行为，而 ON COMPLETION PRESERVE 子句将导致事件不会被丢弃。

ENABLE 和 DISABLE 选项分别表明事件在被创建出来后的初始状态是“启用”（按计划运行）还是“禁用”（不运行）。从 MySQL 5.1.18 版开始新增的 DISABLE ON SALVE 选项的含义是：事件在你创建它的那个 MySQL 服务器上是启用的，但在对它进行复制的任何从服务器上都是禁用的。

在事件创建时生效的 sql_mode 系统变量的值会被保存起来并在事件发生时恢复生效。

事件既不需要输入参数，也不产生输出信息。也就是说，你无法向事件传递参数，而输出信息——例如 SELECT 语句生成并通常返回到客户端的结果集——将被丢弃。

CREATE EVENT 语句是从 MySQL 5.1.6 版开始引入的。DEFINER 子句是从 5.1.17 版开始引入的。

- CREATE FUNCTION、 CREATE PROCEDURE

```
CREATE
  [DEFINER = definer_name]
  FUNCTION routine_name ([func_param [, func_param] ...])
  RETURNS type
  [characteristic] ...
  routine_stmt

CREATE
  [DEFINER = definer_name]
  PROCEDURE routine_name ([proc_param [, proc_param] ...])
  [characteristic] ...
  routine_stmt

func_param:
  param_name type

proc_param:
  [IN | OUT | INOUT] param_name type

characteristic:
    [NOT] DETERMINISTIC
  | LANGUAGE SQL
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT 'str'
```

这些语句用来创建新的存储函数和存储过程。从 MySQL 5.0.3 版开始，你必须具备 CREATE ROUTINE 权限才能使用这些语句来创建存储例程。

在默认的情况下，新例程将被创建在当前的默认数据库里。如果想在某个数据库里创建一个新例程，就应该以 *db_name.routine_name* 的格式来给出它的名字。在同一个数据库里，任意两个函数或两个过程不允许有同样的名字，但函数可以和过程有同样的名字。

存储函数的参数是通过给出参数名及其类型的方式定义的。参数类型可以是任何一种合法的 MySQL 数据类型。参数的作用是在函数被调用时把参数值传递给它，但参数值的变化在函数返回时对调用者而言是不可见的（换句话说，函数的参数都是些 IN 参数）。

在定义一个函数的时候，在参数列表的后面必须加上一条 RETURN 语句以表明其返回值的数据类型。

存储过程的参数也是通过给出参数名及其类型的方式定义的，但在参数名的前面还可以加上 IN、OUT 或 INOUT 限定符以表明该参数是仅用于输入、仅用于输出，还是可以同时用于输入和输出。

- ❑ IN 参数用来把一个值传递给存储过程。这种参数可以在存储过程里修改，但在存储过程执行结束后从调用者程序里看不出任何改变。
- ❑ OUT 参数不是用来向存储过程传递值的，它在存储过程里的初始值是 NULL，允许在存储过程里修改。OUT 参数的最终结果在存储过程执行结束后对调用者程序而言是可见的。
- ❑ INOUT 参数兼具上述两方面特点，可以用来把一个值传递给存储过程，其最终结果在存储过程执行结束后对调用者程序而言也是可见的。

如果没有给出这些关键字中的任何一个，该参数默认为一个 IN 参数。

可选的 *characteristic* 值由一个或多个以空格分隔的下列选项构成。

- ❑ DETERMINISTIC、 NOT DETERMINISTIC

  DETERMINISTIC 的含义是该函数在你使用同样的参数值去调用它的时候将总是产生同样的结果，而 NOT DETERMINISTIC 的含义则是它不一定总是如此。DETERMINISTIC 选项从 MySQL 5.0.44/5.1.21 版开始由查询优化器使用，在那以前不是这样。
- ❑ LANGUAGE SQL

  表明该存储例程的语言。就目前而言，MySQL 会在解析这个选项后丢弃它。SQL 语言是 MySQL 唯一支持的存储例程语言，所以这个指令不是必需的。不过，如果有可能把某个存储例程移植到另外一个支持多种语言的数据库系统上去，你或许想用一条 LANGUAGE 指令来明确地表明它是用 SQL 语言编写的。
- ❑ SQL SECURITY

  这个选项的作用是配合 DEFINER 子句在存储例程执行时确定其信息安全上下文（即用来核查其访问权限的账户），详细情况请参阅 4.5 节。如果 DEFINER 子句没有给出，MySQL 将默认使用当初执行 CREATE 语句的那个用户账户来执行该函数/过程。（这也是在最早引入 DEFINER 子句的 5.0.20/5.1.8 版之前所采取的策略。）从 MySQL 5.0.3 版开始，只有拥有某个存储例程的 EXECUTE 权限的账户才能调用该例程。存储例程的创建者将自动获得该例程的 EXECUTE 和 ALTER RUOTINE 权限（这种授权行为可以通过禁用 automatic_sp_privileges 系统变量来关闭）。
- ❑ COMMENT

  给例程加上一个描述性注释。这种注释可以通过 SHOW CREATE FUNCTION、SHOW CREATE PROCEDURE、SHOW FUNCTION STATUS 和 SHOW PROCEDURE STATUS 等语句查看。

routine_stmt 部分是由 SQL 语句构成的例程主体，应该只包含一条 SQL 语句。如果需要使用多条语句，必须用关键字 BEGIN 和 END 把它们括起来以构成一个复合语句（请参阅 E.2 节）。

函数要向调用者返回一个值，所以在函数体内必须包含至少一条 RETURN 语句。不过，函数无法执行那些会生成一个结果集的语句。

在例程创建时生效的 sql_mode 系统变量的值会被保存起来并在例程执行时恢复生效。

- CREATE INDEX

```
CREATE [UNIQUE | FULLTEXT | SPATIAL] INDEX
  index_name [index_type]
  ON tbl_name (index_columns) [index_option] ...

index_type: USING {BTREE | HASH | RTREE}

index_option:
    index_type
  | COMMENT 'str'
```

```
  | KEY_BLOCK_SIZE [=] n
  | WITH PARSER parser_name
```

给 *tbl_name* 数据表增加一个名为 *index_name* 的新索引。新索引由 *index_columns* 列表里给出的数据列构成，该列表由一个或多个以逗号分隔的数据列构成。CREATE INDEX 语句在 MySQL 内部是被当做 ALTER TABLE 语句来处理的，详细情况请参阅 ALTER TABLE 语句条目。（如果需要为某个数据表创建多个索引，最好使用 ALTER TABLE 语句。只用一条语句就可以添加所有的索引，这比一个一个地创建要快得多。）

UNIQUE、FULLTEXT 或 SPATIAL 关键字可以用来表明索引的类型。如果没有给出它们当中的任何一个，创建的将是一个非唯一化索引。CREATE INDEX 语句不能用来创建 PRIMARY KEY，而必须使用 ALTER TABLE 语句来创建。

FULLTEXT 和 SPATIAL 索引只适用于 MyISAM 数据表。FULLTEXT 索引只适用于非二进制字符串数据列（CHAR、VARCHAR、TEXT），SPATIAL 索引只适用于具备 NOT NULL 属性的空间数据列。

有些存储引擎还允许为新索引指定索引算法，即上述语法中的 *index_type* 部分。算法值可以是 BTREE（适用于 MyISAM 和 InnoDB 数据表）、HASH 或 BTREE（适用于 MEMORY 数据表）和 RTREE（适用于 MyISAM 数据表里的 SPATIAL 索引）。

在 MySQL 5.0.60/5.1.10 版之前，如果想给出 *index_type* 子句，就必须让它出现在 ON *tbl_name* 子句的前面。那以后的 MySQL 版本对这个位置不再有硬性要求，*index_type* 子句应该作为一个 *index_option* 值在索引定义的结尾部分给出。

在 MySQL 5.0 系列版本里（从 5.0.60 版开始），唯一允许出现在索引定义的结尾部分的 *index_option* 值就是 *index_type* 子句（如前所述）。在 MySQL 5.1 系列版本里（从 5.1.10 版开始），允许使用的 *index_option* 值包括 *index_type* 子句和下列选项。

- COMMENT '*str*'新索引提供一个描述性注释（最多 1 024 个字符）。这个选项在 MySQL 5.2.4 及以后的版本里都可以使用。
- KEY_BLOCK_SIZE [=] *n* 建议存储引擎使用 *n* 个字节作为新索引的键块长度。*n* 值为 0 时表示使用默认长度。
- WITH PARSER *parser_name* 只适用于 FULLTEXT 索引，其作用是为新索引指定一个解析器插件。对解析器插件的详细介绍见《MySQL 参考手册》。

有关索引创建方面的更多信息，请参阅 2.6.4 节。

- CREATE SREVER

```
CREATE SERVER server_name
  FOREIGN DATA WRAPPER wrapper_name
  OPTIONS (option [, option] ...)

option:
    USER 'str'
  | PASSWORD 'str'
  | HOST 'str'
  | PORT n
  | DATABASE 'str'
  | SOCKET 'str'
  | OWNER 'str'
```

在定义一个用来访问某远程 MySQL 数据表的 FEDERATED 数据表时，你必须定义连接到那台远程服务器的途径。办法之一是按照如下所示的语法使用一个数据表 CONNECTION 选项明确地列出必要

的连接参数：

```
CONNECTION =
  'mysql://user_name[:password]@host_name[:port_num]/db_name/tbl_name'
```

另一个办法就是使用 CREATE SERVER 语句，该语句将创建一个服务器定义，也就是在 mysql.server 数据表里创建一个数据行来存放连接参数。执行 CREATE SERVER 语句需要具备 SUPER 权限。有了那个服务器定义，就可以在定义 FREDERATED 数据表时在 CONNECTION 选项里引用该定义而无须给出连接字符串了。如果有多个 FREDERATED 数据表需要使用同一组连接参数，一个服务器定义可以大大简化这些数据表的创建工作。

本语法中的 *server_name* 部分是你将要创建的服务器定义的名字，你在定义一个 FREDERATED 数据表时将在其 CONNECTION 选项里引用它。CONNECTION 选项可以按以下格式之一给出（对于第二种格式，远程数据表的名字与本地数据表的名字应该是相同的）：

```
CONNECTION = 'server_name/remote_table_name'
CONNECTION = 'server_name'
```

服务器的名字最长可以有 64 个字符，并且不区分字母大小写。这种名字的有效范围是本地服务器的全局范围，所以 mysql.servers 数据表里的每一个服务器定义都必须有一个独一无二的名字。

wrapper_name 值应该是字符串 mysql，加引号或不加引号均可。

OPTION 子句负责给出连接参数，它的每个选项值必须是一个字符串常数或一个数值常数，这一点在 CREATE SERVER 语句的语法描述里体现得很明显。如果某个字符串选项或数值选项没有给出，其默认值将分别是空字符串或 0。字符串选项最长可以有 64 个字符，数值选项则必须大于或等于 0。

服务器定义里的 *wrapper_name* 值和 OPTION 子句的大部分选项值与相应的连接字符串 'mysql://...'所明确给出的各个连接参数分别对应。OWNER 选项将被保存到 mysql.servers 数据表里，但目前没有实际用途。

这个语句是从 MySQL 5.1.15 版开始引入的。

- CREATE TABLE

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name
  {
      (create_definition,...) [table_option] ...
        [partition_scheme] [trailing_select]
    | [(create_definition,...) [table_option] ...
        [partition_scheme] trailing_select
    | LIKE tbl_name2
    | (LIKE tbl_name2)
  }

table_option: (see following discussion)

trailing_select:
  [IGNORE | REPLACE] [AS] select_stmt

create_definition:
    col_name col_definition [reference_definition]
  | [CONSTRAINT [name]] PRIMARY KEY
      [index_name] [index_type]
      (index_columns) [index_option] ...
  | [CONSTRAINT [name]] UNIQUE [INDEX | KEY]
      [index_name] [index_type]
      (index_columns) [index_option] ...
```

```
  | {INDEX | KEY} [index_name] [index_type]
      (index_columns) [index_option] ...
  | {FULLTEXT | SPATIAL} [INDEX | KEY]
      [index_name] (index_columns) [index_option] ...
  | [CONSTRAINT [name]] FOREIGN KEY [fk_name]
      (index_columns) [reference_definition]
  | CHECK (expr)

col_definition:
  data_type
    [NOT NULL | NULL] [DEFAULT default_value]
    [AUTO_INCREMENT] [PRIMARY KEY] [UNIQUE [KEY]]
    [COMMENT 'str']

index_type: (see following discussion)

index_option: (see following discussion)

reference_definition:
  REFERENCES tbl_name (index_columns)
    [ON DELETE reference_action]
    [ON UPDATE reference_action]
    [MATCH FULL | MATCH PARTIAL | MATCH SIMPLE]

reference_action:
  RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

partition_scheme:
  PARTITION BY
    {
        RANGE(expr)
      | LIST(expr)
      | [LINEAR] HASH(expr)
      | [LINEAR] KEY(col_list)
    }
    [PARTITIONS n]
    [SUBPARTITION BY
      {
          [LINEAR] HASH(expr)
        | [LINEAR] KEY(col_list)
      }
      [SUBPARTITIONS n]
    ]
    [(partition_definition [, partition_definition] ...)]

partition_definition:
  PARTITION partition_name
    [VALUES {LESS THAN {(expr) | MAXVALUE} | IN (value_list)}]
    [partition_option] ...
    [(subpartition_definition [, subpartition_definition] ...)]

subpartition_definition:
  SUBPARTITION subpartition_name
    [partition_option] ...

partition_option: (see following discussion)
```

CREATE TABLE 语句将在默认数据库里创建一个名为 *tbl_name* 的新数据表来。但如果数据表的名字是以 *db_name.tbl_name* 的形式给出的，该数据表就将创建在指定的数据库里。这条语句的执行

需要具备数据表的 CREATE 权限。

一般说来，如果打算创建的数据表已经存在，这条语句将执行失败并报告出错。但这里又有两个例外。首先，如果给出了 IF NOT EXISTS 子句，数据表将不会被创建，这条语句也不会报告出错。其次，如果给出了 TEMPORARY 关键字而已经存在的那个同名数据表不是一个临时数据表，这条语句就会创建出一个临时数据表来。在这个临时数据表存在期间，原来那个名为 *tbl_name* 的数据表将自动“隐藏”起来而不让创建该临时数据表的那个客户（程序）看到，但其他客户（程序）仍能看到原来的那个数据表，因为临时表只有它的创建者看得见。如果你用 DROP TABLE 语句丢弃了这个临时数据表或者把它重新命名为另外一个名字，就可以再次看到原已存在的那个数据表了。必须具备 CREATE TEMPORARY 权限才能创建临时表。

如果给出了 TEMPORARY 关键字，临时数据表将在当前客户连接结束（不管是正常结束还是非正常结束）或者你用 DROP TABLE 语句丢弃它的时候自动“消失”。

create_definition 部分是你准备在这个数据表里创建的各数据列和索引的名称，但如果新数据表是通过尾缀 SELECT 语句而创建出来的，这个部分就可以省略。*table_ options* 子句可以为新数据表设定各种属性。如果新数据表是通过尾缀 *select_stmt* 部分（它可以是任意形式的 SELECT 查询语句）而创建出来的，新数据表将使用那个 SELECT 语句所返回的结果集来创建。关于这几个子句的详细讨论见随后的那几个小节。

数据列和索引的定义。CREATE TABLE 语句的 *create_definition* 部分可以是一个数据列或索引定义、一个 FOREIGN KEY 子句或者一个 CHECK 子句。MySQL 在遇到 CHECK 子句时会解析它但会忽略它，FOREIGN KEY 子句也大致如此，但 InnoDB 数据表上的 FOREIGN KEY 子句除外。

数据列定义将以一个数据类型 data_type 开头，后面通常还会有几个可选的关键字，类型可以是附录 B 里列出的任何一种。每种数据列类型都有一些特有的属性，详细讨论请参见附录 B。允许出现在数据类型之后的其他可选关键字如下所示。

- NULL 或 NOT NULL

  用来表明该数据列是否允许包含 NULL 值。如果这两个关键字都没有给出，则以 NULL 为默认设置。

- DEFAULT *default_value*

  用来设定该数据列的默认值。注意，不存在为 BLOB、TEXT、空间类型或有 AUTO_INCREMENT 属性的数据列设置默认值的问题。除了 TIMESTAMP，数据列的默认值必须是一个常数，它可以是一个数值、一个字符串或者 NULL 值。如果没有设置默认值，就将由 MySQL 安排一个，相关信息参见 3.2.3 节。

- AUTO_INCREMENT

  这个关键字只能用在整数和浮点类型的数据列上。AUTO_INCREMENT 数据列的特殊之处在于：当你往它插入一个 NULL 值时，实际插入的数据值将是有关序列的下一个编号值，它通常等于该数据列里的当前最大编号值再加上 1。在默认的情况下，AUTO_INCREMENT 数据列的实际取值将从 1 开始编号。（有些存储引擎可以通过数据表选项 AUTO_INCREMENT 来明确地设定一个起始编号值，参见下面对表选项的介绍。）AUTO_INCREMENT 数据列必须有索引，还必须为 NOT NULL。每个数据表最多也只能有一个 AUTO_INCREMENT 数据列。

- [PRIMARY] KEY

  用来表明该数据列是一个 PRIMARY KEY。PRIMARY KEY 数据列必须同时被声明为 NOT NULL。如果省略这个项，MySQL 将向列定义中添加 NOT NULL。

- UNIQUE [KEY]

  用来表明该数据列是一个 UNIQUE 索引。
- COMMENT *'str'*

  给数据列加上一个描述性的注释，可以用 SHOW CREATE TABLE 和 SHOW FULL COLUMNS 语句查看。每个注释最多可以包含 1024 个字符（在 MySQL 5.2.4 版之前是 255 个字符）。

PRIMARY KEY、UNIQUE、INDEX、KEY 、FULLTEXT 和 SPATIAL 等子句的用途是创建各种索引。PRIMARY KEY 和 UNIQUE 子句所创建的索引不允许包含相同的值（也就是所谓的唯一化索引）。INDEX 和 KEY 互为同义词，它们所创建的索引允许包含彼此相同的值（也就是所谓的非唯一化索引）。这几个子句所建立的索引将建立在 *index_columns* 部分所列举出来的数据列上，如果涉及多个数据列，则必须以逗号把它们分隔开。如果没有给出索引名，MySQL 将根据第一个带索引数据列的名字自动选定一个。

FULLTEXT 和 SPATIAL 索引只适用于 MyISAM 数据表，FULLTEXT 索引只适用于非二进制字符串数据列（CHAR、VARCHAR、TEXT），SPATIAL 索引只适用于具备 NOT NULL 属性的空间数据列。

PIMARY KEY 数据列必须具备 NOT NULL 属性，即使在定义 PIMARY KEY 数据列时省略了 NOT NULL 属性，MySQL 也会自动给它加上。

对于带有 *index_type* 或 *index_option* 子句的索引定义，有些存储引擎还允许给出索引算法或其他索引定义限定符。关于不同的 MySQL 版本都允许使用哪些索引选项的细节，请参阅 CREATE INDEX 语句条目。关于索引创建方面的更多信息，请参阅 2.6.4 节。

数据表选项。每个 *table_option* 子句指定下面列出的一个数据表特征。每个选项设置都能应用于所有存储引擎，除非有特殊说明。“＝”是可选的，设置值可用空白或逗号隔开。

- AUTO_INCREMENT = *n*

  为数据表设定一个起始编号值。这个选项只能用在 MyISAM 和 MEMORY 数据表以及从 MySQL 5.0.3 版本开始的 InnoDB 数据表里。对于 InnoDB 表，如果在生成任意 AUTO_INCREMENT 值之前重启服务器，这个项就将失效。
- AVG_ROW_LENGTH = *n*

  数据表中的数据行平均长度。对于 MyISAM 数据表，MySQL 将根据 AVG_ROW_LENGTH 和 MAX_ROWS 的乘积来确定数据文件的最大长度。MyISAM 存储引擎内部使用的数据行指针的宽度可以是 2 到 7 个字节，这个指针的默认宽度足以创建出长度高达 4 GB 的数据表来。如果需要用到更大的数据表（并且你的操作系统也支持如此之大的文件），就可以利用 AVG_ROW_LENGTH 和 MAX_ROWS 选项来调整 MyISAM 存储引擎内部使用的数据行指针的宽度。这两个选项值的乘积越大，存储引擎内部使用的数据行指针的宽度也就越大，文件尺寸高达 65536TB。反之，乘积越小，可使用的指针越小。如果数据表本身的尺寸比较小，这种做法就节省不了多少空间，但如果你有很多小尺寸的数据表，累积起来的节省总量就很可观了。
- [DEFAULT] CHARACTER SET=*charset*

  为数据表指定一个默认字符集。charset 可以是某个字符集的名字，也可以是 DEFAULT，即数据表将使用数据库的当前字符集。如果你在定义字符串数据列时没有明确表明它将使用哪一个字符集，该数据列里的数据值就将使用本选项所确定的字符集。在下面的例子里，数据列 c1 将使用 sjis 字符集，而 c2 则将使用 ujis 字符集：

  ```
  CREATE TABLE t
   (
  ```

```
  c1 CHAR(50) CHARACTER SET sjis,
  c2 CHAR(50)
) CHARACTER SET ujis;
```

这个选项还将影响到你以后使用 ALTER TABLE 语句对字符串数据列的修改操作，如果你没有明确地指定一个字符集，有关的字符串数据列就将使用这个选项所指定的字符集。

- CHECKSUM = { 0 | 1 }

  如果这个选项被设置为 1，MySQL 就将为数据表生成一个校验和，修改数据表时也会修改校验和。校验和会给数据表的更新操作稍微增加一点儿开销，但能提高数据表检查操作的工作效率。这个选项只适用于 MyISAM 数据表。

- [DEFAULT]COLLATE=*collation*

  表的默认字符集排序方式。*collation* 可能是一个排序方式名称，或者为 DEFAULT，表示使用表的字符集的默认排序方式。

- COMMENT='*str*'

  给数据表增加一条描述性注释，你可以通过 SHOW CREATTE TABLE 和 SHOW TABLE STATUS 语句来查看，注释的长度最大为 2048 个字符（在 MySQL 5.2.4 之前是 60 个字符）。

- CONNECTION='*connect_str*'

  向 FEDERATED 表表明如何连接远程服务器。这个选项是在 MySQL 5.0.13 中引入的。旧的版本应该使用 COMMENT 来指定'*connect_str*'。

- DATA DIRECTORY = '*dir_name*'

  这个选项只适用于 MyISAM 数据表和 Unix。它用来规定数据文件（即.MYD 文件）必须写到指定的子目录里去。'*dir_name*'必须是一个完整的路径名。这个选项只能工作在服务器未使用--skip-symlink-links 选项启动的场合里。在某些 Unix 操作系统上，symlink 无法配合线程机制工作，因而通常会被默认地禁用掉。

- DELAY_KEY_WRITE = { 0 | 1 }

  如果这个选项被设置为 1，键缓存里的内容将定期被写到磁盘上而不是在每个插入操作完成之后立刻进行这种写操作。这个选项只适用于 MyISAM 数据表。这能提高性能，但如果发生崩溃，可能需要修复数据库。

- ENGINE = *engine_name*

  为数据表指定一种存储引擎。对各种存储引擎的描述见 2.6.1 节。 如果没有对 MySQL 服务器做过其他配置，默认的存储引擎将是 MyISAM。你可以在启动 MySQL 服务器的时候通过 --default-storage-engine 选项把另一种存储引擎设为默认引擎。给定的 MySQL 服务器所支持的存储引擎可以用 SHOW ENGINES 语句查出。如果在创建某个数据表时为它指定的存储引擎不可用，CREATE TABLE 语句的执行效果将取决于 NO_ENGINE_SUBSTITUTION SQL 模式的设置值。

- INDEX DIRECTORY='*dir_name*'

  这个选项与 DATA DIRECTORY 相似，但指定的是编写索引文件（.MYI）的目录。它受到的限制与 DATA DIRECTORY 相同。

- INSERT_METHOD = { NO | FIRST | LAST }

  这个选项用来设定 MERGE 数据表将如何插入数据行。NO 表示根本不允许插入数据行，FIRST 或 LAST 则表示数据行将被插入到组成这一 MERGE 数据表的第一个或最后一个 MyISAM 数据表。

- KEY_BLOCK_SIZE = *n*

  建议存储引擎使用 *n* 个字节作为索引的键块长度。*n* 值为 0 表示使用默认的长度。如果某个索引定义了自己的 KEY_BLOCK_SIZE 选项，该选项的设置值将取代相应的数据表级默认值。这个选项是从 MySQL 5.0.10 版开始引入的。

- MAX_ROWS = *n*

  向存储引擎表明打算存放到数据表里的数据行的最大个数，可以为很大的数。这个选项的具体用法见上面对 AVG_ROW_LENGTH 选项的介绍。

- MIN_ROWS = *n*

  向存储引擎表明打算存放到数据表里的数据行的最小个数。这个选项主要用于 MEMORY 数据表，它能向 MEMORY 存储引擎提供一些内存优化方面的提示。

- PACK_KEYS = { 0 | 1 | DEFAULT }

  这个选项控制着 MyISAM 数据表中的索引压缩功能，即是否需要对相似的索引值进行压缩。索引压缩功能会增加数据表更新操作的开销，但能改善检索操作的性能。0 表示不压缩，1 表示对字符串（即 CHAR、VARCHAR、BINARY 或 VARBINARY）值以及数值型索引值压缩，DEFAULT 表示只对长字符串数据列进行压缩。

- PASSWORD = '*str*'

  设定一个用来加密数据表格式文件的口令。如果没有与 MySQL 数据库系统的运营方签定服务支持合同，这个选项通常没有什么效果。

- ROW_FORMAT =

  { DEFAULT | DYNAMIC | FIXED | COMPRESSD | REDUNDANT | COMPACT }

  为数据行指定一种存储类型。对于一个 MyISAM 数据表，选项值 DYNAMIC 和 FIXED 分别对应于可变长度和固定长度的数据行格式。选项值 COMPRESSED 只能由 myisampack 程序设置，它表明该数据表是经过压缩和只读的。

  REDUNDANT 和 COMPACT 格式适用于 InnoDB 数据表。MySQL 5.0.3 及其后的版本默认使用 COMPACT 格式。如果仍想使用原先的格式，给出 ROW_FORMAT = REDUNDANT 即可。

  如果这个选项所设定的数据行格式不适用，存储引擎将忽略这个选项。比如说，如果数据表包含 BLOB 或 TEXT 数据列，FIXED 格式显然不适用。SHOW TABLE STATUS 语句的 Row_format 输出列值可以告诉我们存储引擎实际选用的是什么格式。

- TYPE = *engine_name*

  这是 ENGINE 数据表选项的一个已被淘汰的同义词。它目前只在 MySQL 5.2.5 之前的版本里还可以使用，但会导致一条警告消息，在 5.2.5 节之后的版本里已不复存在。

- UNION = (*tbl_list*)

  这个选项只适用于 MERGE 数据表，它列出了构成 MERGE 数据表的各个 MyISAM 数据表（以逗号分隔）。

尾缀的 SELECT 语句。如果 CREATE TABLE 语句包含 *select_stmt* 子句（作为尾缀的 SELECT 查询命令）部分，MySQL 就将使用该子句所返回的结果集来创建新数据表。对于那些会导致唯一化索引出现重复值的数据行，MySQL 将按以下原则处理：如果给出了 IGNORE 关键字，则忽略后出现的数据行；如果给出了 REPLACE 关键字，则后出现的数据行将替换掉先出现的数据行；如果这两个关键字都没有给出，CREATE TABLE 语句将执行失败并报告出错。

尾缀的 LIKE 子句。如果 CREATE TABLE 语句的末尾部分是 LIKE *tbl_name2* 子句，新数据表将

是 *tbl_name2* 数据表的一个空白副本。你必须具备 *tbl_name2* 数据表上的 SELECT 权限。新数据表将包括同样的数据列定义、索引定义和数据表选项，但不会复制 DATA DIRECTORY 和 INDEX DIRECTORY 数据表选项，也不复制外键定义。

外键支持。InnoDB 存储引擎提供了外键支持机制。外键必须通过在子数据表里使用关键字 FOREIGN KEY 进行定义，在这个关键字的后面是一个可选的外键 ID，然后是构成这一外键的数据列清单，最后是一个 REFERENCES 定义。外键 ID 即使给出也会被忽略，除非 InnoDB 自动为外键创建一个索引，也就是说，*fr_name* 将成为索引名称。在外键的 REFERENCES 定义里，需要列出该外键所用到的父数据表和数据列，并规定父表中的记录被删除时应该采取什么动作，默认动作是防止删除或更新父表或子表，这会危害引用的整体性。RESTRICT 和 NO ACTION 操作都表示不指定任何动作。ON DEFAULT 和 ON UPDATE 子句用于指定显式动作。InnoDB 实现的动作是 CASCADE（删除或更新相应的子表数据行）和 SET NULL（把相应的子表数据行中的外键列设置为 NULL）。SET DEFAULT 动作没有实现，InnoDB 会报告出错。

REFERENCE 定义中的 MATCH 子句将被解析然后忽略。如果你是为非 InnoDB 类型的存储引擎里定义外键的，那么整个定义都将被忽略。

对于外键的深入讨论，参见 2.14 节。

数据表分区选项。MySQL 5.1 引入了对数据表分区的支持，我们可以在定义数据表的时候对它进行分区并安排其数据被存储到不同的分区。接下来的讨论将对数据表分区的定义语法进行简要的描述，对这个问题的深入讨论和相关示例请参阅 2.6.2 节的第 6 小节和《MySQL 参考手册》。

对数据表分区的定义以 PARTITION BY 开始，接下来是一个分区函数，该函数用来为数据表里的每一个数据行计算一个值，这些值的作用是确定把数据行存储到哪一个分区。每个分区定义还可以包含下面这些可选的组成部分。

- 一个 PARTITIONS *n* 子句：用来表明数据表有多少个分区。*n* 应该是一个正整数。如果给出了这个子句并且还给出了一些 *partition_definition* 子句，*partition_definition* 子句的个数必须等于 *n*。包括子分区在内，每个数据表的最大分区个数是 1024。
- 一条关于如何把分区进一步划分成子分区的描述。
- 一组用来定义各个分区的 *partition_definition* 子句，每个 *partition_definition* 子句各自定义了一个分区的特性，除了为分区提供的一个名字，还可以包括一个用来描述都有哪些分区函数值将被映射到该分区的 VALUES 子句、其他分区选项和一组子分区定义。*subpartition definition* 子句和 *partition_definition* 子句很相似，只不过前者描述的是一个子分区并且不允许再包含 VALUES 子句或子分区定义。

  分区函数总共有 4 种，数据表里的数据行将根据其计算结果被分配到不同的分区。在下面的描述里，*expr* 代表由数据表里的一个或多个数据列构成的表达式，*col_list* 则是一个由一个或多个以逗号分隔的数据列的名字构成的列表。数据列的名字只能来自将被创建的那个数据表。
- RANGE（*expr*） 将把每个分区和表达式 *expr* 的可取值范围的一个子集关联在一起。这种分区函数必须和一个包含着 VALUES LESS THAN 子句的分区定义搭配使用，并按照该子句所给出的整数上限把函数值映射到不同的分区。（NULL 值将被映射到第一个分区。）对于首尾相连的分区，它们的 VALUES 子句所给出的上限值必须是按递增顺序列出的。最后一个分区可以使用 MAXVALUES 关键字作为其分区函数值，其含义是所有未落入此前各分区的函数值都将属于这最后一个分区。

```
CREATE TABLE t (income BIGINT, ...)
```

```
PARTITION BY RANGE (income)
(
  PARTITION p0 VALUES LESS THAN (10000),
  PARTITION p1 VALUES LESS THAN (30000),
  PARTITION p2 VALUES LESS THAN (75000),
  PARTITION p3 VALUES LESS THAN (150000),
  PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

- LIST (*expr*) 将把每个分区和一列值关联在一起。这种分区函数必须和一个包含着 VALUES IN 子句的分区定义搭配使用，并按照该子句所给出的整数列表把函数值映射到不同的分区。如果 *expr* 表达式的计算结果可以是 NULL 值，VALUES 列表之一必须包含 NULL 值。

```
CREATE TABLE t (id INT NULL, ...)
PARTITION BY LIST(id)
(
  PARTITION p0 VALUES IN (1, 2, 3),
  PARTITION p1 VALUES IN (4, 5, 6, NULL)
);
```

- HASH (*expr*)将根据以数据行的内容计算出来的 *expr* 值把数据行存储到相应的分区。典型的做法是把 HASH()函数和用来创建多个分区的 PARTITION *n* 子句搭配使用，并根据 *expr* 除以 *n* 的余数把数据行存储到不同的分区。

```
CREATE TABLE t (d DATE, ...)
PARTITION BY HASH(TO_DAYS(d))
PARTITIONS 5;
```

  在 HASH()形式的分区函数前面还可以加上 LINEAR 限定符，这会使得该函数的算法有所变化。使用 LINEAR 限定符的好处之一是可以让某些分区管理操作（例如通过 ALTER TABLE 语句来增加或丢弃分区等操作）变得更有效率，但这有可能会让数据行在各个分区上的分布程度比没有使用 LINEAR 限定符时的情况更差。

- KEY (*col_list*) 的效果类似于 HASH()形式的分区函数，但你可以指定使用数据表里的哪些数据列来计算散列值，而散列函数将由 MySQL 服务器负责提供。在 KEY()分区函数前面也可以加上 LINEAR 限定符。

如果使用了 HASH()或 KEY()函数，与之搭配使用的分区定义就不应该再有 VALUES 子句。VALUES 子句只能与 RANGE()和 LIST()分区函数搭配使用。

expr 表达式必须具备确定性，而这是为了确保同样的输入总是会得到同样的结果。比如说，在 *expr* 表达式里可以使用 ABS()函数，但不允许使用 RAND()函数。如果使用了一个不允许使用的函数，CREATE TABLE 语句将返回一条出错消息。

对于 RANGE()或 LIST()形式的分区函数，*expr* 表达式的求值结果必须是一个整数或 NULL 值。对于 HASH()形式的分区函数，*expr* 表达式的求值结果必须是一个非 NUUL 非负的整数，所以，如果表达式引用了任何非整数列，它必须将列值转换为整数。比如说，如果 d 是一个 DATE 数据列，你可以使用 TO_DAYS(d)函数把日期转换为天数以保证 HASH(TO_DAYS(d))是一个符合要求的分区函数。

至于 KEY()形式的分区函数，它的参数是数据列的名字，但这些数据列不必是整数类型。

每种 *partition_option* 值对应着一种关于数据表分区的附加特性，它们可以从下面选择。（虽然在以下描述里使用的术语是“分区”，但这些操作也可以用在子分区的定义里。）下列各设置项里的等号“=”是可选的。

- COMMENT = '*str*'

给分区加上一个描述性的注释。

- DATA DIRECTORY = '*dir_name*', INDEX DIRECTORY = '*dir_name*'
  这两个选项类似于前面描述过的同名的数据表选项。它们分别用来设定该分区里的数据或索引将被存储到什么地方。默认的存储位置是该数据表所在数据库的目录。
- MAX_ROWS = *n*、 MIN_ROWS = *n*
  这些选项用来表明你计划让该分区容纳数据行的最大值和最小值。*n* 必须是一个正整数。
- [STORAGE] ENGINE = *engine_name*
  用来处理该分区的存储引擎。数据表分区不支持混合使用多种存储引擎，所以必须为同一个数据表的所有分区指定同一种存储引擎。

以下语句演示了 CREATE TABLE 语句的一些用法。

- 创建一个包含 3 个数据列的数据表。id 数据列是一个 PRIMARY KEY，在 last_name 和 first_name 数据列上还有一个多数据列索引：

```
CREATE TABLE customer
(
  id          SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
  last_name   CHAR(30) NOT NULL,
  first_name  CHAR(20) NOT NULL,
  PRIMARY KEY (id),
  INDEX (last_name, first_name)
);
```

- 创建一个临时数据表。为了提高访问速度，把它创建为一个 MEMORY 数据表：

```
CREATE TEMPORARY TABLE tmp_table
  (id MEDIUMINT NOT NULL UNIQUE, name CHAR(40))
  ENGINE = MEMORY;
```

- 创建一个新数据表作为另一个数据表的空白副本：

```
CREATE TABLE prez_copy LIKE president;
```

- 用另一个数据表的内容创建一个新数据表：

```
CREATE TABLE prez_copy SELECT * FROM president;
```

- 用另一个数据表的部分内容创建一个新数据表：

```
CREATE TABLE prez_alive SELECT last_name, first_name, birth
  FROM president WHERE death IS NULL
```

对于通过使用尾缀 SELECT 语句而创建和填充的数据表，那些定义将在表内容已被插入到该数据表里以后生效。比如说，你可以像下面这样把一个选定的数据列定义为新数据表里的 PRIMARY KEY：

```
CREATE TABLE new_tbl (PRIMARY KEY (a)) SELECT a, b, c FROM old_tbl;
```

你还可以对新数据表里的数据列进行定义以覆盖那些基于结果集特性的默认定义：

```
CREATE TABLE new_tbl
(a INT UNSIGNED NOT NULL AUTO_INCREMENT, b DATE, PRIMARY KEY (a))
  SELECT a, b, c FROM old_tbl;
```

- CREATE TRIGGER

```
CREATE
  [DEFINER = definer_name]
```

```
TRIGGER trigger_name trigger_time trigger_event
ON tbl_name
FOR EACH ROW trigger_stmt
```

把一个触发器和一个数据表关联在一起，当该数据表上发生特定的事件时，该触发器将被激活并执行它所包含的语句。在默认的情况下，*tbl_name* 数据表来自当前的默认数据库。如果针对的是某个特定的数据库里的一个数据表，必须按照 *db_name.tbl_name* 的格式来给出它的名字。

从 MySQL 5.1.6 版开始，CREATE TRIGGER 语句需要你在与触发器相关联的数据表上具备 TRIGGER 权限才能使用；在 5.1.6 版之前，你必须具备 SUPER 权限。

当触发器被激活时，DEFINER 子句将被用来确定其信息安全上下文（即用于核查其访问权限的账户），请参阅 4.5 节。比较新的 MySQL 版本默认使用当初执行 CREATE TRIGGER 语句的那个用户账户来执行触发器，在 MySQL 5.0.17 版之前（DEFINER 子句最早出现于这个版本），MySQL 将使用因为执行某语句而激活触发器的那个用户账户来执行。相关账户必须同时具备目标数据表上的 TRIGGER 权限（在 MySQL 5.1.6 版之前是必须具备 SUPER 权限）、*tbl_name* 数据表上的 SELECT 权限（如果在触发器的定义里使用 NEW 或 OLD 语法引用了 *tbl_name* 数据表里的数据列）和 *tbl_name* 数据表上的 UPDATE 权限（如果在触发器的定义里使用 SET NEW.*col_name* 子句对 *tbl_name* 数据表里的任何数据列进行了修改）。在此基础上，该账户还必须具备为了正常执行触发器定义里的语句而必不可少的各项权限。

trigger_time 值应该是 BEFORE 或 AFTER，分别表示应该在激活触发器的那条语句对每个数据行进行处理之前或之后执行触发器所包含的语句。

trigger_event 值应该是 INSERT、UPDATE 或 DELETE，用来表明哪种语句将导致触发器被激活。

trigger_stmt 部分是由 SQL 语句构成的触发器主体。它应该只包含一条 SQL 语句。如果需要使用多条语句，必须用关键字 BEGIN 和 END 把它们括起来以构成一个复合语句。（参见 E.2 节。）

在 DELETE 或 UPDATE 触发器里，可以用 OLD.*col_name* 语法来引用将被删除或更新的“老”数据行里的数据列。类似地，在 INSERT 或 UPDATE 触发器里，可以用 NEW.*col_name* 语法来引用将被插入或更新的“新”数据行里的数据列。OLD 和 NEW 关键字都不区分字母的大小写。

在 BEFORE 触发器里，可以使用 SET 语句去修改新数据行里的值，如下所示：

```
SET NEW.col_name = value
```

在触发器创建时生效的 sql_mode 系统变量的值会被保存起来并在触发器被激活时恢复生效。

触发器不带任何参数，并且像存储函数那样，无法执行会产生一个结果集的语句。

CREATE TRIGGER 语句是从 MySQL 5.0.2 版开始引入的。

- CREATE USER

```
CREATE USER account [IDENTIFIED BY [PASSWORD] 'password']
  [, account [IDENTIFIED BY [PASSWORD] 'password'] ] ...
```

创建一个或多个 MySQL 账户，每个账户将导致 mysql.user 数据表增加一个没有任何权限的数据行。如果某个账户已经存在，你将看到一条出错消息。账户名必须以'user_name'@ 'host_name'的格式给出，详见 12.4.1 节的第 1 小节。

如果给出了 IDENTIFIED BY 子句，它将给新账户设置一个口令。PASSWORD 关键字通常可以省略，此时只需要以明文的形式给出口令的文本值就行了。不要使用 PASSWORD()函数，用 SET PASSWORD 语句设置口令时才需要那么做。如果某些特殊的场合需要你以 PASSWORD()函数返回值的格式来给出口令的加密值，在该值的前面加上 PASSWORD 关键字即可（当你利用 SHOW GRANT 语句的输出内容去

重新创建一个账户时就需要这么做，这是因为 SHOW GRANT 语句的输出内容里只有经过加密的口令值，没有其明文形式。）

这条语句是从 MySQL 5.0.2 版开始引入的，你必须具备全局级 CREATE USER 权限或 mysql 数据库上的 INSERT 权限才能使用它。

- CREATE VIEW

```
CREATE [OR REPLACE]
  [ALGORITHM = {MERGE | TEMPTABLE | UNDEFINED}]
  [DEFINER = definer_name]
  [SQL SECURITY = {DEFINER | INVOKER}]
  VIEW view_name [(col_list)] AS select_stmt
  [WITH [CASCADED | LOCAL] CHECK OPTION]
```

创建一个视图。如果已经存在一个同名的视图，你将看到一条出错消息，除非你还给出了 OR REPLACE 子句（此时，新视图将取代老视图）。

如果给出了 *col_list* 部分，新视图将由该列表里的数据列构成，而新视图里的每个数据列都必须名列其中。如果没有给出 *col_list* 部分，新视图将由其视图定义里的 SELECT 语句所选取的数据列构成。

select_stmt 是一条用来定义视图的 SELECT 语句，在该语句里可以引用数据表或其他视图。

要想创建一个视图，你必须具备相应的 CREATE VIEW 权限、*select_stmt* 语句所选取的各数据列上的相应权限以及在 *select_stmt* 语句中的其他地方引用的每个数据列上的 SELECT 权限。如果使用了 OR REPLACE 子句，你还必须具备老视图上的 DROP 权限。

在视图被调用的时候，DEFINER 和 SQL SECURITY 子句将被用来确定其信息安全上下文（即用于核查其访问权限的账户），请参阅 4.5 节。默认情况是使用当初执行 CREATE VIEW 语句的那个用户账户。

ALGORITHM 子句决定着如何处理视图：如果被设置为 MERGE，当你发出一条引用了该视图的语句时，MySQL 将把该视图的定义合并到那条语句里，然后执行合并出来的语句；如果被设置为 TEMPTABLE，MySQL 在用到该视图的时候将把它创建为一个临时数据表；如果被设置为 UNDEFINED，MySQL 服务器将根据具体情况自行选择如何处理。默认设置是 UNDEFINED。

WITH CHECK OPTION 子句作用于可更新视图（“可更新视图”是指那些可以配合 UPDATE 或其他数据表修改语句来修改其底层数据表的视图）。它使我们可以利用视图对底层数据表进行数据行插入或修改操作，而如此插入或修改的数据行必须满足视图定义里的 SELECT...WHERE 子句条件。CASCADE 和 LOCAL 关键字在视图定义引用其他视图时起作用。CASCADE 关键字作用于底层视图；LOCAL 关键字的含义则只限于当前视图。如果没有任何一个，MySQL 将默认使用 CASCADED 关键字。

CREATE VIEW 语句是从 MySQL 5.0.1 版开始引入的。WITH CHECK OPTION 子句是从 MySQL 5.0.2 版开始引入的。DEFINER 和 SQL SECURITY 子句最早实现于 5.0.16 版。

- DEALLOCATE PREPARE

```
{DEALLOCATE | DROP} PREPARE stmt_name
```

释放此前使用 PREPARE 语句进行预处理而得到的那条名为 *stmt_name* 的预处理语句。被释放的预处理语句将不再能够执行。

- DELETE

```
DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name
  [WHERE where_expr] [ORDER BY ...] [LIMIT n]

DELETE [LOW_PRIORITY] [QUICK] [IGNORE] tbl_name[.*] [, tbl_name[.*]] ...
```

```
    FROM tbl_refs
    [WHERE where_expr]

  DELETE [LOW_PRIORITY] [QUICK] [IGNORE] FROM tbl_name[.*] [, tbl_name[.*]] ...
    USING tbl_refs
    [WHERE where_expr]
```

DELETE 语句的第一种形式用来从数据表 *tbl_name* 里把那些符合 WHERE 子句所给条件的数据行全部删除掉。第二种和第三种形式可以从多个表中删除行，或者根据有关多表的条件删除行。tbl_refs 的语法与 SELECT 类似，但你不能把一个子查询指定为一个表。

```
DELETE FROM score WHERE event_id = 14;
DELETE FROM member WHERE expiration < CURDATE();
```

如果 WHERE 子句被省略，则数据表中的全体数据行都将被删除！

LOW_PRIORITY 选项（如果给出的话）将使 DELETE 语句被延迟到没有客户程序读该数据表时才执行。这个选项只对使用表级别锁定的存储引擎有效，如 MyISAM、MEMORY 和 MERGE。

对于 MyISAM 数据表，给出 QUICK 选项将加快语句的执行速度，MyISAM 存储引擎将不执行通常要执行的索引树页结点合并工作。

如果使用了 IGNORE 修饰符，在数据行被删除时发生的错误将被忽略，但会生成警告。

LIMIT 子句（如果给出的话）将把该 DELETE 语句将要删除的数据行的最大个数限制为 n。

ORDER BY 子句（如果给出的话）将使 DELETE 语句先对结果集排序，然后再进行删除操作。把它与 LIMIT 子句结合使用，就能更精确地控制将要删除哪些数据行。ORDER BY 子句的语法与 SELECT 语句相同。

在一般情况下，DELETE 语句会把它实际删除的数据行个数作为返回值。但不带 WHERE 子句的 DELETE 语句会清空数据表，这种做法速度很快，但返回的数据行计数值却可能是 0。如果想知道到底删除了多少行，就需要给出一条能够匹配全体数据行的 WHERE 子句，如下所示：

```
DELETE FROM tbl_name WHERE TRUE;
```

但这种逐行删除的做法将会大大降低性能。

如果不需要知道到底删除了多少行，还可以利用 TRUNCATE TABLE 语句来迅速清空整个数据表。

DELETE 语句的第二、第三种形式特别适用于从多个数据表一次性删除有关数据行。它们使你能够根据表之间的联结关系来确定需要删除哪些行。在这两种形式的 DELETE 语句里，数据表的名字既可以写成 *tbl_name* 的形式，也可以写成 *tbl_name*.*（这种写法是为了与 ODBC 保持兼容）的形式。

tbl_refs 子句用来给出为了判断应该删除哪些行而需要关联的数据表。在这个子句里，你可以在引用数据表时为它们声明一个别名。DELETE 语句的其他部分可以引用、但不能声明数据表别名。

如果想从 t1 数据表里删除 id 值与 t2 数据表里的 id 值相匹配的那些行，可以使用如下所示的第一种多数据表语法：

```
DELETE t1 FROM t1 INNER JOIN t2 WHERE t1.id = t2.id;
```

或者使用如下所示的第二种语法：

```
DELETE FROM t1 USING t1 INNER JOIN t2 WHERE t1.id = t2.id;
```

多数据表 DELETE 语句不允许有 ORDER BY 或 LIMIT 子句，也不允许其 WHERE 子句里的子查询（如果有的话）从数据行将被删除的那个数据表选取数据行。

● DESCRIBE

```
{DESCRIBE | DESC} tbl_name [col_name | 'pattern']
```

```
{DESCRIBE | DESC} select_stmt
```

DESCRIBE 语句的第一种形式，带数据表名称或视图名称（自 MySQL 5.0.1 起），将产生与 SHOW COLUMNS 语句完全相同的输出报告，详情请参见后面的 SHOW 条目。在这种语法里，如果给定数据列，DESCRIBE 语句的输出报告中就将只包含关于该数据列的信息；如果你还给出了一个字符串（'pattern'），DESCRIBE 语句就会把它视为一个匹配模式（如 LIKE 操作符），它的输出报告里就将包含所有与这个模式相匹配的数据列的有关信息。

❑ 下面这条语句显示关于 president 数据表里的 last_name 数据列的描述信息：

```
DESCRIBE president last_name;
```

❑ 下面这条语句显示关于president数据表里的last_name和first_name两个数据列的描信息：

```
DESCRIBE president '%name';
```

DESCRIBE 语句的第二种形式（带一个 SELECT 查询语句）其实是 EXPLAIN 语句的同义词，详细情况请参见后面的 EXPLAIN 条目。（MySQL 里的 DESCRIBE 和 EXPLAIN 语句其实是功能完全相同的同义词，但人们通常习惯于使用 DESCRIBE 语句来获得数据表的描述信息、使用 EXPLAIN 语句来获得关于 SELECT 查询语句的描述信息。）

● DO

```
DO expr [, expr] ...
```

对表达式求值，但不返回求值结果。因为不需要处理结果集，所以 DO 语句比 SELECT 语句更适合对表达式求值。DO 语句可用来设置变量，或者调用那些你只关心其“副作用”而不关心返回值的函数，如下所示：

```
DO @sidea := 3, @sideb := 4, @sidec := SQRT(@sidea*@sidea+@sideb*+@sideb);
DO RELEASE_LOCK('mylock');
```

● DROP DATABASE

```
DROP DATABASE [IF EXISTS] db_name
```

即删除指定的数据库及其内容。如果打算删除的数据库不存在（除非你给出了 IF EXISTS 关键字）或者不具备 DROP 权限，这条语句将执行失败。IF EXISTS 的作用是抑制（即不显示）MySQL 在试图删除一个并不存在的数据库时给出出错信息，但这种情况下会生成警告。

一个数据库就是数据目录的一个子目录。服务器只删除它认为已自动创建的文件和目录（如.frm）文件。而这个子目录里的非数据表文件不会被 DROP DATABASE 删除。这会导致数据库目录删除失败，而且 DROP DATABASE 执行失败。在这种情况下，SHOW DATABASES 输出仍会列出数据库。为了解决这个问题，需要手动删除所有的额外文件和子目录，然后再发出 DROP DATABASE 语句。

DROP DATABASE 语句在执行成功后将返回一个计数值，表示实际删除的数据表或视图的个数（这个计数值其实是它实际删除的.frm 文件的个数，但二者的含义是一样的）。

● DROP EVENT

```
DROP EVENT [IF EXISTS] event_name
```

删除名为 *event_name* 的事件。如果省略了 IF EXISTS 子句，试图删除一个并不存在的事件将导致 MySQL 生成一条出错消息；如果带有 IF EXISTS 子句，MySQL 将生成一条警告消息。你必须具

备给定数据库的 EVENT 权限才能删除该数据库里的事件。(在 MySQL 5.1.12 版之前，必须具备 SUPER 权限或者是该事件的定义者。)

DROP EVENT 语句是从 MySQL 5.1.6 版开始引入的。

- DROP FUNCTION、DROP PROCEDURE

```
DROP {FUNCTION | PROCEDURE} [IF EXISTS] routine_name
```

删除名为 *routine_name* 的存储函数或存储过程。

如果省略了 IF EXISTS 子句，试图删除一个并不存在的例程将导致 MySQL 生成一条出错消息；如果带有 IF EXISTS 子句，MySQL 将生成一条警告消息。

从 MySQWL 5.0.3 版开始，必须具备给定例程的 ALTER ROUTINE 权限才能删除它。

- DROP INDEX

```
DROP INDEX index_name ON tbl_name
```

从数据表 *tbl_name* 里删除名为 *index_name* 的索引。MySQL 将把这条语句当做一条 ALTER TABLE DROP INDEX 语句来处理，详情请参见前面的 ALTER TABLE 条目。若要使用 DROP INDEX 语句删除一个 PRIMARY KEY，必须用一个标识符来引述后者。

```
DROP INDEX `PRIMARY` ON tbl_name;
```

- DROP SERVER

```
DROP SERVER [IF EXISTS] server_name
```

通过从 mysql.servers 表中移除相应行来删除名为 *server_name* 的 FEDERATED 表服务器的定义，但必须具备 SUPER 权限才能执行。这条语句是从 MySQL 5.1.15 引入的。

- DROP TABLE

```
DROP [TEMPORARY] {TABLE | TABLES} [IF EXISTS] tbl_name [, tbl_name] ...
  [RESTRICT | CASCADE]
```

将指定数据表从所在数据库中删除，如果指定 TEMPORARY 关键字，只删除 TEMPORARY 表。

如果给出 IF EXISTS 子句，当你试图删除的数据表并不存在时，不会像通常那样报告出错，但会生成警告。

- DROP TRIGGER

```
DROP TRIGGER [IF EXISTS] db_name.trigger_name
```

从指定数据库删除触发器，语句中需要包含数据库名字。

如果给出 IF EXISTS 子句，当你试图删除的触发器并不存在时，不会像通常那样报告出错，但会生成警告。

如果一个表有触发器，那么删除这个表也会删除它的触发器。

DROP TRIGGER 是从 MySQL 5.0.2 开始引入的。从 MySQL 5.1.6 开始，需要具备触发器关联的数据表的 TRIGGER 权限才能执行。在 5.1.6 之前，则必须具备 SUPER 权限。IF EXISTS 子句是从 MySQL 5.0.32/5.1.14 开始引入的。

- DROP USER

```
DROP USER account [, account] ...
```

从 MySQL 5.0.2 开始，DROP USER 子句删除与账户相关的权限表的所有数据行，这会删除账户及其权限。

在 MySQL 5.0.2 之前，DROP USER 只删除没有权限的账户和与账户相关联的 mysql.user 数据表行。为了完整地删除账户，首先要使用 SHOW GRANTS 来查看账户拥有的权限，使用 REVOKE 来调用那些权限，然后发出 DROP USER 语句。

将每个账户按'*user_name*'@'*host_name*'格式命名，参见 12.4.1 节的第 1 小节。如果账户不存在，将会报告出错。

需要具备全局 CREATE USER 权限或者是 mysql 数据库的 DELETE 权限才能执行 DROP USER 子句。

DROP USER 不会删除任何数据库或被删除账户创建的其他对象。

● DROP VIEW

```
DROP VIEW [IF EXISTS] view_name [, view_name] ...
  [RESTRICT | CASCADE]
```

将给定视图从其所在数据库中删除，必须具备这个视图的 DROP 权限才能执行这个子句。

如果给出 IF EXISTS 子句，当你要删除的视图并不存在时，不会像通常那样报告出错，但会生成一个警告。

RESTRICT 和 CASCADE 关键字会被解析但被忽略，它们不会起什么作用。

DROP VIEW 是从 MySQL 5.0.1 开始引入的。

● EXECUTE

```
EXECUTE stmt_name [USING @var_name [, @var_name] ...]
```

执行之前用 PREPARE 预处理过的名为 *stms_name* 的预处理语句。如果预处理语句包含任何占位符标记，那么必须给出 USING 子句。这个子句应该提供一个用逗号分隔的用户变量列表，为子句中每个后续的占位符提供值。

● EXPLAIN

```
EXPLAIN tbl_name [col_name | 'pattern']
```

```
EXPLAIN [EXTENDED | PARTITIONS] select_stmt
```

这条语句的第一种形式相当于一条 DESCRIBE *tbl_name* 语句，详细情况请参见前面的 DESCRIBE 条目。

EXPLAIN 语句的第二种形式能够让我们了解到 MySQL 将如何执行出现在关键字 EXPLAIN 之后的那条 SELECT 语句，如下所示：

```
EXPLAIN SELECT score.* FROM score INNER JOIN grade_event
ON score.event_id = grade_event.event_id AND grade_event.event_id = 14;
```

EXTENDED 选项将使 EXPLAIN 语句生成更多关于执行计划的信息，这些信息可以通过在 EXPLAIN 语句执行完毕后立刻执行一条 SHOW WARNINGS 语句去查看。PARTITIONS 选项（始见于 MySQL 5.1.5 版）将使 EXPLAIN 语句多生成一个关于数据表分区情况的输出列。

如果 *select_stmt* 的 FROM 子句里包含一个子查询，EXPLAIN 语句必须执行那个子查询。这是因

为优化器必须知道那个子查询返回才能为外层查询制订执行计划。

EXPLAIN 语句的输出由一个或多个包含以下输出列的输出行构成。

- id

  这个输出行所对应的 SELECT 语句的 ID 编号。有些语句——例如包含子查询或是使用了 UNION 语法的语句——可以有多个 SELECT 子句。

- select_type

  这个输出行所对应的 SELECT 语句的类型如表 E-1 所示。

表　E-1

类　型	含　义
SIMPLE	一条没有UNION或子查询部分的SELECT语句
PRIMARY	最外层或最左侧的SELECT语句
UNION	UNION语句里的第二条或最后一条SELECT子句
DEPENDENT UNION	和UNION类型的含义相似，但需要依赖于某个外层查询
UNION RESULT	一条UNION语句的结果
SUBQUERY	子查询中的第一个SELECT子句
DEPENDENT SUBQUERY	和SUBQUERY类型的含义相似，但需要依赖于某个外层查询
DERIVED	FROM子句里的子查询

- table

  各输出行里的信息是关于哪个数据表的。

- Partitions

  将要使用的分区。只有出现 PARTITIONS 选项时才显示这列。对于非分区表，这个值为 NULL。

- type

  MySQL 将进行的联结操作的类型。这些类型（从优到劣）包括：system、const、eq_ref、ref、ref_or_null、index_merge、unique_subquery、index_subquery、range、index 和 ALL。排列在前面的类型有着更强的限制性，MySQL 在检索过程中检查的数据行相对少一些。

- possible_keys

  MySQL 认为在指定数据表（即名称出现在 table 列里的那个数据表）里对数据行进行检索时可能用到的索引。如果这个输出列里的值是 NULL，则表明没有找到索引。

- key

  MySQL 在指定数据表里对数据行进行检索时实际用到的各有关索引的名字。（如果 MySQL 使用了 index-merge 联结类型，这里可能列出好几个键，因为优化器使用好几个索引来处理查询。）如果这个输出列里的值是 NULL，则表明没有在该数据表里找到这样的索引。

- key_len

  实际使用的索引的长度。如果 MySQL 实际使用的是索引的最左前缀（即只使用了单个索引项的前 *n* 个字节），这个数字可能会小于单个索引项的总长度。

- ref

  MySQL 用来与索引值比较的值，如果是单词 const 或'???'，则表示比较对象是一个常数；如果是某个数据列的名称，则表示比较操作是逐个数据列进行的。

- rows

  MySQL 为完成查询而需要在数据表里检查的行数的估算值。这个输出列里的值的乘积就是所有数据表中必须检查的数据行的各种可能组合的估算值。

- Extra

  有关执行计划的其他信息。这个值可以为空，或包含一个或多个如下所示的值。
  - Using filesort：需要将索引值写到文件中并且排序，这样按顺利检索相关数据行。
  - Using index：MySQL 可以不必检查数据文件，只使用索引信息就能检索数据表信息。
  - Using temporary：必须创建的临时表。
  - Using where：利用 SELECT 语句中的 WHERE 子句里的信息进行索引操作。

此外，还可能出现这里没列出的其他值，参见《MySQL 参考手册》了解当前所有的 Extra 值。

- FLUSH

```
FLUSH [LOCAL | NO_WRITE_TO_BINLOG] option [, option] ...
```

对 MySQL 服务器内部使用的各种缓存进行刷新。下面是这条语句的 *option* 部分的可取值：

- DES_KEY_FILE

  重新加载供 DES_ENCRYPT()和 DES_DECRYPT()函数用来完成加密/解密工作的 DES 密钥文件。

- HOSTS

  刷新主机缓存里的信息。

- LOGS

  刷新日志文件：先关闭这些文件，再重新打开它们。如果启用了二进制日志或中继日志，LOGS 将导致后面的文件打开。对于记录到文件中的错误，老文件会被重命名，有一个_old 后缀。

- MASTER

  这个选项现已被重新命名为 RESET MASTER，请使用新名字。

- PRIVILEGES

  重新加载权限数据表。如果通过 GRANT 或 REVOKE 命令修改了这些数据表，MySQL 服务器将自动同步更新它们在内存中的复制；但如果是通过 INSERT 或 UPDATE 等语句来直接修改权限数据表，就必须利用这个选项明确地让 MySQL 重新加载它们。这个选项对账户资源管理的限制类似于 USER_RESOURCES 选项。

- QUERY CACHE

  刷新查询缓存以对之进行碎片整理,但不清除这个缓存里的语句。(如果想彻底清除这个缓存，就需要使用 RESET QUERY CACHE 命令。)

- SLAVE

  这个选项已被重新命名为 RESET SLAVE。

- STATUS

  重新对状态变量进行初始化。

- {TABLE | TABLES } [*tbl_name [, tbl_name]*...]

  如果你没有给出任何一个数据表的名字，将关闭数据表缓存里所有已经打开的数据表。如果你以逗号分隔列出了一个或多个数据表的名字，将只刷新你指定的那几个数据表而不是整个数据表缓存。

  如果查询缓存里有值得“刷新”的东西，FLUSH TABLES 语句还将刷新查询缓存区。

- ❑ TABLES WITH READ LOCK

  先刷新所有数据库里的所有数据表，然后给它们加上一个读操作锁，这个读操作锁将一直作用到你发出一条 UNLOCK TABLES 语句为止。这条语句仍允许客户（程序）读取数据表里的内容，但将阻塞对这些数据表的修改（写）操作。如果想把服务器的内容完整地备份下来，使用本选项将确保数据表不会在你备份操作期间发生任何改变。当然，从客户的角度看，不利的是，这意味着无法修改数据表的时间被延长了。

- ❑ USER_RESOURCES

  把当前账户的各项资源每小时配额（比如 MAX_QUERYS_PER_HOUR）重置为它们各自的初始值，这样，那些资源使用量已经达到配额上限的账户又可以使用了。这个选项不会影响 MAX_USER_CONNECTIONS 的限制，它不是对每小时配额的限制。

  如果启用了二进制日志，MySQL 会把 FLOSH 语句写到二进制日志里，除非给出了 LOCAL 或 NO_WRITE_TO_BEGIN 选项。

  FLUSH 语句需要有 RELOAD 权限才能执行。

- GRANT

```
GRANT priv_type [(col_list)] [, priv_type [(col_list)] ] ...
  ON [TABLE | FUNCTION | PROCEDURE]
    {*.* | * | db_name.* | db_name.tbl_name | tbl_name | db_name.routine_name}
  TO account [IDENTIFIED BY [PASSWORD] 'password']
    [, account [IDENTIFIED BY [PASSWORD] 'password'] ] ...
  [REQUIRE security_options]
  [WITH grant_or_resource_options]
```

GRANT 语句将把访问权限授予一位或者多位 MySQL 账户。要使用这条语句，必须具备 GRANT OPTION 权限和将要授予的那个权限。

每个 *priv_type* 值都指定一个要授予的权限，如表 E-2 所示。除必须单独使用的 ALL 权限外，必须用逗号把权限隔开。ALL 是所有其他权限的组合，但 GRANT OPTION 必须单独授予或通过增加 WITH GRANT OPTION 子句授予。

表 E-2

权限名称	该权限允许的操作
ALTER	更改数据表和索引的定义
ALTER ROUTINE	更改或丢弃存储函数和过程
CREATE	创建数据库和表
CREATE ROUTINE	创建函数和过程
CREATE TEMPORARY TABLES	使用TEMPORARY关键字创建临时表
CREATE USER	使用高级账户管理语句
CREATE VIEW	创建视图
DELETE	从表中删除行
DROP	删除数据库、表和其他对象
EVENT	创建、丢弃和更改事件调度程序
EXECUTE	执行存储函数和过程
FILE	读、写服务器主机上的文件
GRANT OPTION	把本账户的权限授予其他账户

（续）

权限名称	该权限允许的操作
INDEX	创建或删除索引
INSERT	往表中插入新行
LOCK TABLES	用LOCK TABLES语句显式锁定数据表
PROCESS	查看服务器上运行的线程相关的信息
REFERENCES	未使用（供以后使用）
RELOAD	重加载权限表或刷新日志和或缓存
REPLICATION CLIENT	查知主从服务器的地址
REPLICATION SLAVE	作为复制机制中的从服务器运行
SELECT	检索表中的行
SHOW DATABASES	用SHOW DATABASE语句查看所有数据库名称
SHOW VIEW	用SHOW CREATE VIEW语句查看视图定义
SHUTDOWN	关闭服务器
SUPER	终止线程，执行其他超级用户操作
TRIGGER	创建或丢弃触发器
UPDATE	修改数据行
ALL [PRIVILEGES]	所有操作（GRANT除外）
USAGE	一个特殊的“无权限”权限

LOCK TABLES 权限只能作用于你还拥有 SELECT 权限的那些数据表，但它允许放置任何种类的数据锁，而不仅限于读操作锁。

你总是能查看或者终止自己的线程，PROCESS 或 SUPER 权限允许你查看或终止任何账户的线程。

CREATE VIEW 和 SHOW VIEW 权限最早出现于 MySQL 5.0.1。ALTER ROUTINE 和 CREATE ROUTINE 最早出现于 MySQL 5.0.3，只应用于存储例程，而不是用户定义的函数（UDF）。在 MySQL 5.0.3 中，引入了 CREATE USER 和 EXCUTE 权限。EVENT 和 TRIGGER 最早出现于 MySQL 5.1.6。（在 5.1.6 之前，操作触发器需要 SUPER 而不是 TRIGGER。）

ON 子句负责设定权限的作用范围，如表 E-3 所示。

表 E-3

权限限定符	权限的作用范围
ON *.*	全局级权限，作用于所有数据库、所有表
ON *	如果未指定默认数据库，则为全局级权限；否则，为数据库级权限
ON *db_name*.*	数据库级权限，作用于给定数据库的所有对象
ON *db_name*.*tbl_name*	数据表级权限，作用于给定表中所有列
ON *tbl_name*	数据表级权限，作用于默认数据库给定表中所有列
ON *db_name*.*routine_name*	作用于给定数据库中给定例程的权限

从 MySQL 5.0.6 版开始，如果担心出现歧义，可以加上 TABLE、FUNCTION 或 PROCEDURE 关键字来明确地表明你正在授予的权限是作用于哪种数据库对象的（例如 ON TABLE mydb.mytbl 或 ON FUNCTION mydb.myfunc）。

当你使用 ALL 作为权限名时，只有当前授权级别上的可用权限会被授予对方。比如说，RELOAD 权限是一个全局级权限，所以只有在 ON *.*级别上使用 GRANT ALL 语句才会把它授予对方，在 ON *db_name*.*级别上则不行。在后一种情况里，实际授予的只有数据库级别的全部权限。ALL 还可以用来授予全局级、数据库级、数据表级或例程级的全部权限。

USAGE 权限的含义是“没有任何权限”，它应该只在全局级别使用。

GRANT OPTION 作用于给定级别的所有权限。比如说，你无法把某给定数据库的 SELECT 和 INSERT 权限授予某个账户，但可以让该账户把其中某个权限授予其他人的。

如果某个数据表的名字出现在了 ON 子句里，还可以在权限后面加上一条（*col_list*）子句，把该权限进一步授予到数据列，（*col_list*）子句由一个或多个以逗号分隔的数据列的名字构成。（这只适用于 INSERT、REFERENCE、SELECT 和 UPDATE 权限，只有这几种权限允许授予数据列。）

在为数据表或数据列授权时，相应的数据表或数据列必须已经存在。

TO 子句用来列出将获得权限的一个或多个账户。请使用'*user_name*'@'*host_name*'的格式来给出每个账户，详见 12.4.1 节的第 1 小节。在每个账户名的后面还可以加上一条可选的 IDENTIFIED BY 子句来设定口令。

如果打算给数据库、数据表、数据列和例程的名字加上引号，必须使用标识符引号字符。用户名和主机名则既可以使用标识符引号字符，也可以使用字符串引号字符。例如：

```
GRANT INSERT ('mycol') ON 'test'.'t' TO 'myuser'@'localhost';
```

如果给出 IDENTIFIED BY 子句，它将给被授权账户设置一个口令，相关语法见 CREATE USER 条目里的描述。如果被授权账户已经存在，GRANT 语句的 IDENTIFIED BY 子句将把该账户的老口令替换为新口令；否则，该账户的现有口令将保持不变。

如果被授权账户尚不存在，GRANT 语句将创建它。为了避免在使用 GRANT 语句的时候意外地创建出一个没有口令的新账户（这是不安全的），应该启用 NO_AUTO_CREATE_USER SQL 模式。这个模式在 MySQL 5.0.2 及以后的版本里都可以使用，其作用是阻止不带 IDENTIFIED BY 子句的 GRANT 语句创建一个新账户。

如果给出 REQUIRE 子句，它将要求被授权账户必须使用安全连接，并指定客户应提供的信息。在 REQUIRE 关键字的后面可以给出以下选项。

- NONE。不要求安全连接。
- SSL。常用的连接类型，指定账户必须通过 SSL 来连接 MySQL 服务器。
- X509。指定账户必须提供一份合法的 X509 证书，对 X509 证书的内容没有具体要求，只要合法就行。
- 以下一个或者多个选项。这几个选项对连接和 X509 证书提出了具体的要求。
  - CIPHER '*str*'。指定账户必须以字符串'*str*'作为连接期间的加密密钥。
  - ISSUER '*str*'。证书的签发者必须是'*str*'。
  - SUBJECT '*str*'。证书的主题必须是'*str*'。

如果需要给出一个上述选项，可以用 AND 把它们分隔开（这个 AND 分隔符是可选的），先后顺序无关紧要。

WITH 子句（如果给出的话）规定指定账户是否可以把自己的权限转授给其他用户，并对账户的资源消耗作出限制。WITH 子句的各种选项如下所示，你可以同时使用多个选项，它们的先后顺序无关紧要。

- GRANT OPTION。“转授权”权限，即允许这个账户把自己的权限——包括“转授权”权限在

内——转授给其他账户。

- ❑ MAX_CONNECTIONS_PER_HOUR *n*。这个账户每小时内最多允许建立 *n* 个连接。
- ❑ MAX_QUERIES_PER_HOUR *n*。这个账户每小时可以发出 *n* 条语句。
- ❑ MAX_UPDATES_PER_HOUR *n*。这个账户每小时可以发出 *n* 条修改语句。
- ❑ MAX_USER_CONNECTIONS *n*。被授权账户最多可以和服务器同时建立 *n* 个连接。这个选项是从 MySQL 5.0.3 版开始引入的。

如果 MAX_CONNECTIONS_PER_HOUR、MAX_QUERYS_PER_HOUR 和 MAX_UPDATES_PER_HOUR 选项的设置值是 0，其含义是“没有限制”。如果 MAX_USER_CONNECTIONS 选项的设置值是 0，其含义是“以系统变量 max_user_connections 的设置值为准”。

下面是一些演示 GRANT 语句各种用法的示例，12.4.2 节还有许多这方面的示例。13.3 节讨论了 SSL 的使用。下面都没有给出 IDENTIFIRE，是因为我们假定已使用 CREATE USER 创建了用户，并设置了口令。

- ❑ 让名为 paul 的账户可以从任何一台主机访问 sampdb 数据库里的所有数据表。下面两条语句是等价的，因为账户名中缺失的主机名部分相当于“%”：

```
GRANT ALL ON sampdb.* TO 'paul';
GRANT ALL ON sampdb.* TO 'paul'@'%';
```

- ❑ 允许名为 lookup 的账户从 xyz.com 域里的任何一台主机以只读方式去访问 menagerie 数据库里的数据表：

```
GRANT SELECT ON menagerie.* TO 'lookonly'@'%.xyz.com';
```

- ❑ 允许名为 member_mgr 的账户拥有 sampdb 数据库里 member 数据表上的全部权限（但仅此而已），而且必须从指定主机去连接服务器：

```
GRANT ALL ON sampdb.member TO 'member_mgr'@'boa.snake.net';
```

- ❑ 授予超级用户权限，他可以做任何事，包括向其他用户授权，但只能从本地主机连接服务器：

```
GRANT ALL ON *.* TO 'superduper'@'localhost' WITH GRANT OPTION;
```

- ❑ 允许一位匿名用户访问 menagerie 数据库：

```
GRANT ALL ON menagerie.* TO ''@'localhost';
```

- ❑ 允许账户访问 privatedb 数据库，但必须通过 SSL 连接，而且必须提供一份合法的 X509 证书：

```
GRANT ALL ON privatedb.* TO 'paranoid'@'%.mydomain.com' REQUIRE X509;
```

- ❑ 下面这条 GRANT 语句所创建的账户每小时只能进行 100 次数据库查询，而在这些查询中，只有 10 次允许是数据修改操作：

```
GRANT ALL ON test.* TO 'caleb'@'localhost'
  WITH MAX_QUERIES_PER_HOUR 100 MAX_UPDATES_PER_HOUR 10;
```

- HANDLER

```
HANDLER tbl_name OPEN [[AS] alias_name]

HANDLER tbl_name READ
  {FIRST | NEXT}
  [where_clause] [limit_clause]

HANDLER tbl_name READ index_name
  {FIRST | NEXT | PREV | LAST | < | <= | = | => | >} (expr_list)
  [where_clause] [limit_clause]
```

```
HANDLER tbl_name CLOSE
```

HANDLER 语句提供一个面向 MyISAM 和 InnoDB 存储引擎的底层接口，它可以绕过优化器直接访问数据表内容。如果想通过 HANDLER 接口访问某个数据表，首先要用 HANDLER...OPEN 语句去打开它。这个数据表在你发出 HANDLER...CLOSE 语句（明确地关闭这个数据表）或者结束本次连接之前将一直保持打开状态。在数据表处于打开状态时，可以用 HANDLER...READ 语句访问它的内容。

HANDLER 接口没有提供任何针对数据并发修改情况的保护措施。它不锁定数据表，所以用 HANDLER 语句打开的数据表不会阻塞其他客户程序对其修改，而这些改变不一定能反映在你从表里读出来的记录里。

- INSERT

```
INSERT [DELAYED | LOW_PRIORITY | HIGH_PRIORITY] [IGNORE] [INTO]
  tbl_name [(col_list)]
  {VALUES|VALUE} (expr [, expr] ...) [, (...)] ...
  [ON DUPLICATE KEY UPDATE col_name=expr [, col_name=expr] ...]

INSERT [DELAYED | LOW_PRIORITY | HIGH_PRIORITY] [IGNORE] [INTO]
  tbl_name SET col_name=expr [, col_name=expr] ...
  [ON DUPLICATE KEY UPDATE col_name=expr [, col_name=expr] ...]

INSERT [LOW_PRIORITY | HIGH_PRIORITY] [IGNORE] [INTO]
  tbl_name [(col_list)]
  {SELECT ... | (SELECT ...)}
  [ON DUPLICATE KEY UPDATE col_name=expr [, col_name=expr] ...]
```

把数据行插入到一个已经存在的数据表 *tbl_name* 里，并返回实际插入的行数。INSERT 语法有 3 种形式。

INSERT 语句的第一种形式要求把待插入的数据行全都写在 VALUES()部分里。如果给出了 *col_list*，VALUES()必须为表中每列指定一个值。如果 *col_list* 部分由以逗号分隔的一个或者多个数据列名称构成，每列的值必须在 VALUES()中指定，而没有列出的数据列将被设置为它们的默认值。可以指定多个值 列表，即允许你使用一条 INSERT 语句插入多行。

```
INSERT INTO absence (student_id, date) VALUES(14,'2008-11-03'),(34,NOW());
```

INSERT 语句的 *col_list* 和 VALUE()部分允许同时为空，即允许你往数据表里插入一条各数据列全部是其默认值的数据行，如下所示：

```
INSERT INTO t () VALUES();
```

INSERT 语句的第二种形式把 SET 子句生成的数据列插入到数据表里，SET 子句负责把数据列设置为相应的表达式的值，没有出现在 SET 子句里的数据列将被设置为默认值。

```
INSERT INTO absence SET student_id = 14, date = '2008-11-03';
INSERT INTO absence SET student_id = 34, date = NOW();
```

DEFAULT 可以用在 VALAUES 里或 SET 子句里，将列设置为默认值（在不知道默认值是什么的情况下）。为了在表达式中引用列的默认值，可以使用 DEFAULT(*col_name*)。如果默认值为 NULL，用下列语句将列 i 设置为 0；否则，设置为 1。

```
INSERT INTO t SET i = IF(DEFAULT(i) IS NULL,1,0);
```

INSERT 语句的第三种形式先执行 SELECT 语句，然后再把检索到的记录插入到数据表 *tbl_name*

里。数据列的个数必须等于 *tbl_name* 里的列数或者 *col_list* 部分所列举的列数（如果有 *col_list* 部分）。如果有 *col_list* 部分，那些没有出现的数据列将被设置为默认值。

```
INSERT INTO score (student_id, score, event_id)
  SELECT student_id, 100 AS score, 15 AS event_id FROM student;
```

你不能使用一个子查询从你插入的数据表里选取数据行。

如果数据列在定义时没有 DEFAULT 子句，那么当你在严格 SQL 模式下使用一条 INSERT 语句时，省略该数据列或是使用 DEFAULT 关键字都会导致一个错误。

如果一个将被插入的数据行会导致某个唯一化索引出现重复的键值，INSERT 语句将中止执行并报告出错，剩余的数据行将不再被插入。加上 IGNORE 关键字将使它忽略这样的数据行，继续执行，不会报告出错。在严格 SQL 模式下，IGNORE 关键字还将使得 INSERT 语句把会导致它中止执行的数据转换错误当做一个不那么致命的警告来处理——把导致出错的数据列设置为最接近的合法值。

ON DUPLICATE KEY UPDATE 子句用于解决新插入的数据行导致某个唯一化索引发生键值重复的冲突问题。带有这个子句的 INSERT 语句在遇到上述问题时将被转换为一条 UPDATE 语句，它将使用 UPDATE 关键字后面的数据列赋值表达式对老数据行的数据列进行修改。如果确实发生了一次这样的修改，INSERT 语句返回的受影响数据行的计数值将会是 2，不是 1。

DELAYED、LOW_PRIORITY 和 HIGH_PRIORITY 选项会对 INSERT 操作的实际发生时间产生影响。

- ❑ DELAYED。把新数据行放入一个队列，排队等待插入，INSERT 语句则立刻返回。这种安排的好处是客户无须等待就可以继续进行其他操作，但代价是 LAST_INSERTED_ID() 函数将无法为数据表里的任何一个 AUTO_INCREMENT 数据列返回其 AUTO_INCREMENT 值。DELAYED 插入适用于 MyISAM、MEMORY、ARCHIVE 和（从 MySQL 5.1.19 版开始）BLACKHOLE 数据表。DELAYED 选项在 INSERT INTO...SELECT 语句和 INSERT INTO...ON DUPLICATE KEY UPDATE 语句里将被忽略。DELAYED 选项在以下两种与存储过程或触发器搭配使用的情况里也将被忽略：(1) INSERT 语句是通过某个存储过程去访问数据表或触发器的；(2) INSERT 语句是在某个存储函数或触发器的内部被调用的。
- ❑ LOW_PRIORITY。让 INSERT 语句延迟到没有任何客户正在读取该数据表的时候才实际执行。
- ❑ HIGH_PRIORITY。为当前 INSERT 语句撤销服务器选项--low-priority-updates 的设置效果。（如果在启动服务器时使用了--low-priority-updates 选项，它将降低 INSERT 和其他数据表修改语句的优先级。）HIGH_PRIORITY 选项的另一种用途是预防 INSERT 语句与访问同一个的数据表的 SELECT 语句并发执行。

LOW_PRIORITY 和 HIGH_PRIORITY 选项只适用于 MyISAM、MEMORY 和 MERGE 等具备数据表级锁定功能的存储引擎。

- KILL

```
KILL [CONNECTION | QUERY] thread_id
```

终止 *thread_id* 指定的服务器线程。必须拥有 SUPER 权限才能终止不属于你本人的线程。KILL 语句每次只能终止一个线程，而能够完成同样操作的 mysqladmin kill 命令则允许在命令行上同时给出多个线程的 ID 编号。

CONNECTION 选项的效果和没有任何选项时一样：结束有着给定 ID 的线程。QUERY 选项将结束给定线程正在执行的所有语句，但不结束该线程本身。

- LOAD DATA

```
LOAD DATA [LOW_PRIORITY | CONCURRENT ] [LOCAL] INFILE 'file_name'
```

```
[IGNORE | REPLACE]
INTO TABLE tbl_name
[CHARACTER SET charset]
[field_options] [line_options]
[IGNORE n LINES]
[(col_or_user_var_name, ...)]
[SET col_name = expr, ...]
```

LOAD DATA 语句读出文件 *file_name* 里的记录并把它们批量加载到数据表 *tbl_name* 里去，这要比使用一组 INSERT 语句来完成的速度快。

LOAD DATA 语句将返回一个格式如下所示的信息字符串：

```
Records: n  Deleted: n  Skipped: n  Warnings: n
```

如果 Warning 计数值不是零，请使用 SHOW WARNINGS 语句去查看到底出了什么问题。

LOW_PRIORITY 选项将使 LOAD DATA 语句被延缓到没有客户程序读取该数据表时才执行。LOW_PRIORITY 选项仅适用于使用表级别锁定的存储引擎，如 MyISAM、MEMORY 和 MERGE。

CONCURRENT 选项只适用于 MyISAM 数据表。如果表的中间没有空闲块，新行就会加载到表的末尾。其他客户可以在你往数据表里加载数据时检索该数据表。

如果 LOAD DATA 语句不带 LOCAL 关键字，就将由服务器在服务器主机上直接读取数据文件。这一操作要求你本人必须具备 FILE 权限，并且文件必须位于默认数据库的目录中或者是完全可读的。如果带 LOCAL 关键字，就将由客户程序在客户主机上读取数据文件并把其内容通过网络发送给服务器，这一操作不要求你必须具备 FILE 权限。LOCAL 机制可以有选择地被禁用或者启用。如果服务器端禁用了 LOCAL 机制，你就无法在客户端使用这一机制；如果服务器端启用了 LOCAL 机制而客户却默认地禁用了，就需要由你明确地启用它。比如说，如果使用的是 mysql 客户程序，就可以通过 --local-infile 选项去启用 LOCAL 机制。

如果 LOAD DATA 语句不带 LOCAL 关键字，MySQL 服务器将按以下原则去寻找数据文件。

- 如果'*file_name*'是一个绝对路径名，服务器就将从根目录直接查找文件。
- 如果'*file_name*'是一个相对路径名，那就还要看它是否只有一个成分，如果是，服务器就到默认数据库的目录里去寻找文件；如果它包含有多个成分，服务器将从它自己的数据目录开始去寻找文件。

如果 LOAD DATA 语句带 LOCAL 关键字，客户程序将按以下原则去寻找数据文件。

- 如果'*file_name*'是一个绝对路径名，客户程序就从根目录直接查找文件。
- 如果'*file_name*'是一个相对路径名，客户程序就将从你的当前子目录开始去寻找数据文件。

对于 Windows 系统，文件名中的反斜线字符既可以写成一个斜线字符（/），也可以写成双反斜线字符（\\）。

MySQL 从 5.0.19/5.1.6 版开始使用 character_set_filesystem 系统变量所指定的字符集来解析文件名。

在默认的情况下，MySQL 使用 character_set_database 系统变量所指定的字符集来解析文件的内容。从 MySQL 5.0.38/5.1.17 版开始，你可以用 CHARACTER SET 子句明确地给出一种字符集来解析文件的内容（但目前还不能用这个办法来加载 ucs2、utf18 或 utf32 文件）。

如果数据行会导致数据表里的唯一化索引出现重复值，它们将根据 LOAD DATA 语句给出的是 IGNORE 还是 REPLACE 选项而被忽略或替换。如果两个选项都没有给出，LOAD DATA 将报告出错，尚未加载的数据行将被忽略。不过，如果给出了 LOCAL 限定符，文件传输过程将不允许被打断，因此，

如果没有给出 IGNORE 或 REPLACE 选项中的任何一个，默认行为将和给出了 IGNORE 选项时的情况一样。

field_options 和 *line_options* 子句用来设定数据的格式。(这两个子句里的可用选项也适用于 SELECT...INTO OUTFILE 语句的对应子句。) 下面是这两个子句的语法：

```
field_options:
  [FIELDS
    [TERMINATED BY 'str']
    [[OPTIONALLY] ENCLOSED BY 'char']
    [ESCAPED BY 'char' ] ]

line_options:
  [LINES
    [STARTING BY 'str']
    [TERMINATED BY 'str'] ]
```

'*str*'和'*char*'值允许包含表 E-4 中的转义序列来表示特殊字符，这些转义序列应严格按照表中所示的字母大小写情况写出。

表 E-4

转义序列	含　义
\0	NULL（零值字节）
\b	退格
\n	换行符
\r	回车符
\s	空格
\t	制表符
\'	单引号
\"	双引号
\\	反斜线字符
\z	Ctrl-Z（Window EOF字符）

你还可以用十六进制常数来表示任意字符。比如说，LINES TERMINATED BY 0x02 表示各行数据是以 Ctrl-B（ASCII 2）字符结尾的。

如果给出了 FIELDS 关键字，那么至少还要再给出 TERMINATED BY、ENCLOSED BY 和 ESCAPED BY 等子句中的一个。如果给出多个子句，它们的先后顺序是任意的。类似地，如果给出了 LINES 关键字，那么至少还要再给出 STARTING BY 或 TERMINATED BY 子句中的一个。如果给出多个子句，它们的先后顺序可以是任意的。如果同时给出了 FIELDS 和 LINES 关键字，就必须把 FIELDS 部分安排在 LINES 部分的前面。

FIELDS 子句各组成部分的用法如下所示。

- TERMINATED BY 表明数据文件中同一行上的各项数据值的分隔符。
- ENCLOSED BY 表明数据值的引号符，在把数据插入数据表之前，MySQL 会把这个引号符去掉。是否给出 OPTIONALLY 关键字并不影响 ENCLOSED BY 子句的效果。对于输出(即 SELECT...INTO OUTFILE）语句，由 ENCLOSED BY 所设定的字符用来括住各字段值。如果同时给出了 OPTIONALLY 关键字，则只给来自 CHAR 和 VARCHAR 数据列的值添加引号符。

  要在输入字段值中包含 ENCLOSED BY 所设定的引号符，就必须双写这个引号符或者用 ESCAPED BY 所设定的引导符进行转义。否则，引号符就会被解释为字段值的结束标记。对于输出语句，

MySQL 将自动地在字段值中的引号符前面加上一个转义字符。

- ESCAPED BY 表明如何对特殊字符转义。在下面的例子里，假设转义字符是反斜线字符（\）。
  对于输入语句，不带引号的转义序列\N（反斜线字符“\”+字母“N”）将被解释为 NULL；转义序列\0（反斜线字符“\”+ASCII 0）将被解释为一个取值为零的字节；其他转义序列都将被解释为去掉转义字符之后剩余的东西（即无特殊意义）。比如说，不管字段值是不是已经用双引号括了起来，它里面的转义序列“\”都将被解释为一个双引号字符。
  对于输出语句，转义字符将把 NULL 值编码为不带引号的转义序列\N（零值字节为\0），而 ENCLOSED BY 字符和 ESCAPED BY 字符的实例将像字段分隔符和行分隔符那样被加上一个前导的转义字符。但是，如果 ESCAPED BY 字符是一个空字符（即你给出的是 ESCAPED BY ' ' 子句），则不进行转义处理（此时，NULL 写作 NULL，而不是\N）。如果想把转义字符设定为反斜线字符“\”，就必须双写它（即给出一条 ESCAPED BY '\\'子句）。

LINES 子句各组成部分的用法如下所示。

- STARTING BY 表明数据文件中的各行开头的字符(这个值及其前面的所有内容都作为开头部分)。
- TERMINATED BY 表明各行数据结尾的字符。

如果 FIELDS 和 LINES 子句都没有给出，则各字符将分别使用如下所示的默认值：

```
FIELDS
  TERMINATED BY '\t'
  ENCLOSED BY ''
  ESCAPED BY '\\'
LINES
  STARTING BY ''
  TERMINATED BY '\n'
```

也就是说，同一行上的字段将以制表符分隔且不带引号，转义字符将默认为反斜线字符“\”，各行数据将以换行符结尾。

如果 FIELDS 子句所设定的 TERMINATED BY 字符 ENCLOSED BY 字符都是空字符，就表示数据文件将使用固定宽度的数据行格式，各字段值之间也没有分隔符。数据列值将根据该数据列的显示宽度从数据文件里被读出或者——对于输出语句——被写入数据文件。比如说，输入语句将把 VARCHAR(15) 和 MEDIUMINT(5)数据列分别当做一个 15 个字符宽和一个 8 个字符宽的字段读入；而输出语句将把它们分别写为一个 15 个字符宽和一个 8 个字符宽的字段。NULL 值将写为由空格符构成的字符串。(在 MySQL5.0.6 之前，以列数据类型的显示宽度为基础解释固定宽度。)

输入数据文件里的 NULL 值用不带引号的转义序列\N 表示。如果 FIELDS ENCLOSED BY 字符不为空，那么所有的非 NULL 输入值就都将用给定的引号符括起来，而不带引号的单词 NULL 也将被解释为一个 NULL 值。

如果给出了 IGNORE *n* LINES 子句，则输入的前 *n* 行将被丢弃。比如说，如果数据文件的第一行是一个标题栏而你又不想把它放到数据表里去，就需要使用 IGNORE 1 LINES 子句，如下所示：

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl IGNORE 1 LINES;
```

默认情况下，MySQL 将把输入行里的数据依次赋值给数据表中的每一个数据列。如果给出了由一个或者多个以逗号分隔的数据列名称构成的列表，MySQL 将把输入行里的数据依次赋给每一个给定的数据列。名称没有列出的数据列将被设置为默认值。如果输入行提供的数据值的个数少于数据列的个数，那些没有赋值的数据列就被设置为默认值。

在严格 SQL 模式下执行 LOAD DATA 时，如果列定义中没有 DEFAULT 子句，并且它没有赋值，那

么将会报告出错。

从 MySQL 5.0.3 版开始，名称列表允许由数据列名和用户变量名混合构成，还可以给出一个 SET 子句对输入值进行一些必要的处理后再加载到数据表里。比如说，下面这条语句将把第一个输入列加载到数据表里的 col1 列，忽略第二个输入列，把第三个和第四个输入列的和加载到 col2 列，再使用 UUID()函数为 col3 列提供一个即时生成的值：

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
  (col1,@skip,@addend1, @addend2)
  SET col2 = @addend1 + @addend2, col3 = UUID();
```

SET 子句可以包含多个以逗号分隔的赋值表达式。每个赋值表达式的左侧必须是数据表里的一个数据列的名字。MySQL 不允许使用用户变量来读取固定宽度的输入行，因为它无法根据用户变量来确定各数据列的宽度。标量型子查询可以用来提供数据列值，但子查询所查询的数据表和正被加载数据的数据表不允许是同一个。

如果你有一个在 Windows 平台上创建的、以制表符分隔各字段的文本文件，你可以使用默认的列分隔符，但输入行很可能是以“回车加换行”结束的。为了加载这样的文件，需要给出一个不同的行结束符（“\r”代表回车，“\n”代表换行）：

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
  LINES TERMINATED BY '\r\n';
```

有一部分 Windows 程序在创建数据文件时会沿用 MS-DOS 年代的奇怪惯例，在文件末尾加上 Ctrl-Z 字符作为文件结束标记，加载这样的数据文件可能会在数据库里留下一个异常的数据行。要解决这个问题，可以换个没有这种“坏习惯”的程序来创建文件，也可以在加载文件后及时删掉那个异常的数据行。

采用 CSV（Comma-Separated Value，意思是“以逗号分隔的值”）格式的文件以逗号作为字段之间的分隔符，字段本身或许还用双引号括了起来。假设各输入行以换行符结束，用来加载这类数据文件的 LOAD DATA 语句应该是如下所示的样子：

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
  FIELDS TERMINATED BY ',' ENCLOSED BY '"';
```

控制字符可以用十六进制符号来给出。下面这条语句可以用来加载各字段之间以 Ctrl-A（ASCII 1）字符分隔、输入行以 Ctrl-B（ASCII 2）字符结束的数据文件：

```
LOAD DATA LOCAL INFILE 'mytbl.txt' INTO TABLE mytbl
  FIELDS TERMINATED BY 0x01 LINES TERMINATED BY 0x02;
```

- LOAD INDEX INTO CACHE

```
LOAD INDEX INTO CACHE
  tbl_name [[INDEX | KEY] (index_name [, index_name] ...)
    [IGNORE LEAVES]]
  [, tbl_name [[INDEX | KEY] (index_name [, index_name] ...)]
    [IGNORE LEAVES]]...
```

把给定的 MyISAM 数据表的索引加载到分配给该数据表的键缓存。如果没有使用 CACHE INDEX 语句为某个数据表特意分配一个缓存，该数据表的索引将被加载到默认的键缓存。在默认的情况下，所有的索引块都将被加载。如果给出了 IGNORE LEAVES 子句，只有不是索引树中的叶结点的索引块才会被加载。

类似于 CACHE INDEX 语句的情况，虽然 LOAD INDEX INTO CACHE 语句的语法允许只加载个别的

索引，但它目前实际上把一个数据表的所有索引都加载了。

你必须拥有其中列出的每个数据表的 INDEX 权限才能执行 LOAD INDEX INTO CACHE 语句。

LOAD INDEX INTO CACHE 语句生成的输出信息的格式见 CHECK TABLE 语句条目里的相关描述。

关于 MyISAM 在键缓存管理方面的更多信息，请参阅 12.7.2 节。

- LOCK TABLE

```
LOCK {TABLE | TABLES}
  tbl_name [[AS] alias_name] lock_type
  [, tbl_name [[AS] alias_name] lock_type] ...
```

申请数据锁（即对给定的数据表进行锁定，如有必要，则等待到数据锁全都申请到手为止。数据锁类型 lock_type 必须是下列情况之一。

❑ READ [LOCAL]

申请一个读操作锁。这种数据锁将阻塞其他客户对数据表的写操作，但允许其他客户对数据表进行读操作。

READ LOCAL 是 READ 锁的一个变体，是专为并发插入操作而设计的。它只能用在 MyISAM 数据表上，并且要求 MyISAM 数据表不能有任何因删除或更新操作而产生的空闲块。READ LOCAL 允许你明确地锁定一个数据表，仍允许其他客户程序对它进行并发插入操作。（如果数据表内部存在空闲块，这种锁将被视为一个普通的 READ 锁。）

❑ [LOW_PRIORITY]WRITE

申请一个写操作锁。这种数据锁将阻塞其他客户程序对数据表的任何读、写操作。

LOW_PRIORITY WRITE 申请一个低优先级写操作锁。如果在等待期间又有其他客户程序在读取给定数据表，则允许其他客户进行读操作。只有当没有客户程序在对给定数据表进行读操作时，才能获得 LOW_PRIORITY WRITE 锁。

LOCK TABLE 语句会释放你当前保有的全部数据锁。也就是说，如果想锁定多个数据表，就必须用一条 LOCK TABLE 语句把它们全都锁定。客户程序在结束运行时会自动释放它保有的全部数据锁。

LOCK TABLE 语句允许你给待锁定的数据表起一个别名，这样，当你在今后的查询命令里需要用到这个数据表时，就可以用别名来指称它。（如果需要在同一条查询里多次用到同一个数据表，就必须为该数据表的每一次引用分别申请一个数据锁，在必要时锁定别名。所有锁定都必须在同一条语句中请求。）

```
LOCK TABLE student READ, score WRITE, grade_event READ;
LOCK TABLE member READ;
LOCK TABLE t AS t1 READ, t AS t2 READ;
```

如果正在执行事务，LOCK TABLE 会隐式提交。如果使用 START TRANSACTION 启用事务，使用 LOCK TABLE 获取的表锁定就会隐式释放。

- OPTIMIZE TABLE

```
OPTIMIZE [LOCAL | NO_WRITE_TO_BINLOG]
  {TABLE | TABLES} tbl_name [, tbl_name] ...
```

DELTETE、REPLACE 和 UPDATE 等语句会使数据表的内部出现未使用区域，如果数据表有可变长的数据行，情况就更是如此。对于 MyISAM 表，为了消除这些未使用区域，OPTIMIZE TABLE 语句将进行以下操作。

❑ 对数据表进行碎片整理，消除其中的未使用区域，缩小数据表的尺寸。

❑ 把因碎片化而散布在各处的可变长数据行的内容合并在一起，让各数据行的内容无间断地存放

在同一处。

- 如有必要，为索引页面分类。
- 更新数据表的内部统计信息。

OPTIMIZE TABLE 语句与带--check-only-changed、--quick、--sort-index 和--analyze 选项的 myisamchk 客户程序相同。但在使用 myisamchk 客户程序时，必须设法阻止服务器在你检查数据表的过程中访问它们。使用 OPTIMIZE TABLE 语句就不必这么麻烦了，服务器在完成各项检查工作的同时，还会替你阻止其他客户在某个数据表正在被优化时访问它。

对于 InnoDB 数据表，OPTIMIZE TABLE 语句将被映射为一条 LATER TABLE 语句以刷新各 InnoDB 数据表的索引信息和释放聚集索引里的未使用空间。

从 MySQL 5.0.16 版开始，对于 ARCHIVE 数据表，OPTIMIZE TABLE 语句将进行数据表分析并重新压缩它们以减少其存储空间占用量。

必须具备每一个数据表的 SELECT 和 INSERT 权限才能执行 OPTIMIZE TABLE 语句。

如果启用了二进制日志功能而你又没在 OPTIMIZE TABLE 语句里给出 LOCAL 或 NO_WRITE_TO_BINLOG 选项，MySQL 将把 OPTIMIZE TABLE 语句记载到二进制日志文件里。

OPTIMIZE TABLE 语句生成的输出信息的格式见 CHECK TABLE 语句条目里的相关描述。

- PREPARE

```
PREPARE stmt_name FROM {'str' | @var_name}
```

对一条语句进行预处理并把它命名为 *stmt_name*。经过预处理的语句可以用 EXECUTE 语句来执行，用 DEALLOCATE PREPARE 语句来释放。如果已经存在一个同名的预处理语句，PREPARE 语句将先释放那条老语句后再处理新语句。预处理语句的名字不区分字母的大小写情况。

将接受预处理的语句可以使用一个字符串字面量或用户变量来给出。允许使用 PREPARE 语句进行预处理的语句越来越多，最初只有 CREATE TABLE、DELETE、DO、INSERT、REPLACE、SELECT、SET、UPDATE 语句以及绝大多数 SHOW 语句，其他语句都是后来才增加的。你可以从《MySQL 参考手册》查到你正在使用的 MySQL 版本里都包含哪些语句。PREPARE、EXECUTE 和 DEALLOCATE PREPARE 语句不接受预处理。

可以在预处理语句里使用“?”字符作为占位符，然后等到执行它们时再为其中的占位符提供相应的数据值。占位符使得预处理语句可以接受输入参数，你可以每次使用不同的数据值去执行同一条预处理语句。

PREPARE、EXECUTE 和 DEALLOCATE 语句共同构成了 SQL 语言中的预处理语句接口。不要把它们和第 7 章和附录 G 里讨论的二进制 API 混为一谈，后者是 C 语言中的编程接口，在效率方面要更高一些。

- PURGE MASTER LOGS

```
PURGE {MASTER | BINARY} LOGS {TO 'log_name' | BEFORE 'date'}
```

把服务器上早于给定义文件或给定日期（*'CCYY_MM_DD hh:mm:ss'*格式）生成的二进制日志文件全部删除，重置二进制日志的索引文件，使它只列出那些未被删除的日志。当你在主服务器上运行完 SHOW SLAVE STATUS 语句之后，往往需要使用这个语句来查知仍在使用中的日志文件都有哪些。这条语句要求你必须具备 SUPER 权限。

下面这条语句将把二进制日志 binlog.000001 到 binlog.000009（或者更精确地讲，它们当中现仍存在的）全部删除，并使 binlog.000010 成为未被删除的日志文件中的第一个：

```
PURGE MASTER LOGS TO 'binlog.000010';
```

● RELEASE SAVEPOINT

```
RELEASE SAVEPOINT savepoint_name
```

释放当前事务中名为的 *savepoint_name* 的保存点，如果该保存点不存在，则返回一条出错消息。释放还原点不会导致事务被提交或回滚。这条语句是从 MySQL 5.0.3 版开始引入的。

● RENAME TABLE

```
RENAME {TABLE | TABLES} tbl_name TO new_tbl_name
  [, tbl_name TO new_tbl_name] ...
```

对一个或者多个数据表重新命名。这条语句与 ALTER TABLE...RENAME 语句很相似，但 RENAME TABLE 语句能够同时对多个数据表进行重命名并能在命名过程中锁定它们。如果需要确保给定数据表在重命名的过程中不会被其他客户程序修改，就应该使用 RENAME TABLE 语句。

如果重命名的 InnoDB 数据表有其他数据表通过外键关系依赖于它，InnoDB 存储引擎将自动调整那些依赖关系指向重命名后的数据表。

如果重命名的 MyISAM 数据表是某个 MERGE 数据表的组成部分，你必须相应地重新定义那个 MERGE 数据表。

RENAME TABLE 语句不能用来重命名 TEMPORARY 数据表。

从 MySQL 5.0.2 版开始，如果某个数据表上有触发器，在你试图把它重命名到另一个数据库里去时会导致一个错误。

从 MySQL 5.0.14 版开始，RENAME TABLE 语句也可以用于视图，但你不能把视图重命名到另一个数据库里去。

● RENAME USER

```
RENAME USER from_account TO to_account
  [, from_account TO to_account] ...
```

重命名一个或多个 MySQL 账户。RENAME USER 语句将把每个 *from_account* 重命名为相应的 *to_account*。如果 *from_account* 不存在或是某个 *to_account* 已经存在，则报告出错。账户名必须以'*user_name*'@'*host_name*'的格式给出，详见 12.4.1 节的第 1 小节。

这个语句是从 MySQL 5.0.2 版开始引入的。你必须具备全局级 CREATE USER 权限或是 mysql 数据库的 UPDATE 权限才能执行它。

RENAME USER 语句不会把老账户所具备的权限转移给新账户。

● REPAIR TABLE

```
REPAIR [LOCAL | NO_WRITE_TO_BINLOG]
  {TABLE | TABLES} tbl_name [, tbl_name] ... [option] ...
```

这条语句的用途是对受损数据表进行修复。它只能用在 MyISAM、ARCHIVE，以及 MySQL5.1.19 后的 CSV 数据表上，并且你必须具备每个数据表上的 SELECT 和 INSERT 权限。

不带任何选项的 REPAIR TABLE 语句和 myisamchk --recover 命令的数据表修复效果相同。下面是允许使用的 *option* 值及其含义。这些选项全都适用于 MyISAM 数据表，其他存储引擎不一定能使用它们。

- ❑ EXTENDED 执行包括重建索引等工作在内的高级修复。它与运行 myisamchk --safe-recover 命令来修复数据表的情况相似，只是数据表的修复工作将由 MySQL 服务器而不是由外部实用工具程序来进行。

- ❑ QUICK 只对数据表的索引进行快速修复，不涉及数据文件。
- ❑ USE_FRM 利用数据表的定义文件（即.frm 文件）来重新初始化索引文件，并确定其数据文件的内容需要如何解释，然后再依此重新建立索引。如果你弄丢了索引或者它们损坏得无法修复，这个选项可就有用了。然而，这只能作为最后的选择，而且只有当你的 MySQL 当前版本与创建数据表的 MySQL 版本一样时才能使用，否则将对数据表产生很大的危害。

如果启用了二进制日志功能而你又没在 REPAIR TABLE 语句里给出 LOCAL 或 NO_WRITE_TO_BINLOG 选项，MySQL 将把 REPAIR TABLE 语句记载到二进制日志文件里。

- REPLACE

```
REPLACE [LOW_PRIORITY | DELAYED] [INTO]
  tbl_name [(col_list)]
  {VALUES|VALUE} (expr [, expr] ...) [, (...)] ...

REPLACE [LOW_PRIORITY | DELAYED] [INTO]
  tbl_name [(col_list)]
  {SELECT ... | (SELECT ...)}

REPLACE [LOW_PRIORITY | DELAYED] [INTO]
  tbl_name SET col_name=expr [, col_name=expr] ...
```

REPLACE 语句的基本操作与 INSERT 语句很相似，但如果将被插入的数据行会导致数据表里的唯一化索引出现重复键值，MySQL 将先删除原有的那行，然后再插入新行。因此，REPLACE 语句的语法里不存在 INGORE 选项。同样，REPLACE 不支持 ON DUPLICATE KEY UPDATE。详情请参见前面的 INSERT 条目。

如果数据表有多个唯一化索引，就有可能发生一条 REPLACE 语句删除了多行的事情。例如，当新行同时与几个唯一化索引中的值匹配时，MySQL 会先把所匹配的那几个数据行都删除掉，然后再插入新数据行。

REPLACE 语句要求你必须具备给定数据表上的 INSERT 和 DELETE 权限。

- RESET

```
RESET option [, option] ...
```

RESET 语句对日志和缓存信息的影响与 FLUSH 语句中的情况相同。option 值的可用选项及其作用如下所示。

- ❑ MASTER 删除主服务器上现有的二进制日志文件，重新创建一个新文件并把它编号为 000001，再重置二进制日志的索引文件，使它只列出新文件。
- ❑ QUERY CACHE 清除查询缓存，把当前注册在其中的查询命令全部删除掉。如果只想消除查询缓存中的碎片而不想清除它的全部内容，就应该使用 FLUSH QUERY CACHE 语句。
- ❑ 如果是在一个从服务器上，SLAVE 将删除所有现有的中继日志文件，开始一个新文件，“忘记”复制协调工作（即它当前使用的复制二进制日志的文件名和这些文件中的读写位置）。

RESET 语句要求你必须具备 RELOAD 权限。

- REVOKE

```
REVOKE priv_type [(col_list)] [, priv_type [(col_list)] ...]
  ON [TABLE | FUNCTION | PROCEDURE]
    {*.* | * | db_name.* | db_name.tbl_name | tbl_name | db_name.routine_name}
  FROM account [, account] ...

REVOKE ALL [PRIVILEGES], GRANT OPTION
```

```
  FROM account [, account] ...
```

REVOKE 语句用来撤销给定账户的权限。账户名必须以'*user_name*'@'*host_name*'的格式给出，详见 12.4.1 节的第 1 小节。不存在的账户将导致一个错误。

在第一种语法里，对 *priv_type*、*col_list* 和 ON 子句的要求和 GRANT 语句里的一样。要想使用这条语句，必须同时具备 GRANT OPTION 权限和打算撤销的权限。

第二种语法有一个固定的权限清单，但没有 ON 子句。它将撤销所有给定账户的全部权限。第二种语法要求必须具备全局级 CREATE 权限或 mysql 数据库的 UPDATE 权限。

REVOKE 语句不会从 mysql.user 权限数据表删除账户行。这意味着即使把某个账户的权限全都撤销了，该账户也可以用来连接 MySQL 服务器。要想彻底删除某个账户，必须使用 DROP USER 语句（或者以手动方式从 mysql.user 数据表删除对应于该账户的数据行）。

- 撤销 member_mgr 用户对 sampdb 数据库里的 member 数据表进行各种修改的权限：

```
REVOKE INSERT,DELETE,UPDATE ON sampdb.member
  FROM 'member_mgr'@'boa.snake.net';
```

- 撤销本地主机上的匿名用户对 menagerie 数据库里的某个数据表的全部权限：

```
REVOKE ALL ON menagerie.pet FROM ''@'localhost';
```

- ALL 只能撤销除 GRANT OPTION 以外的全部权限。如果想把 GRANT OPTION 权限也撤销，必须明确地这么做：

```
REVOKE GRANT OPTION ON menagerie.pet FROM "@'localhost';
```

- 撤销 superduper@localhost 账户在所有级别上的全部权限：

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM 'superduper'@'localhost';
```

- ROLLBACK

```
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]

ROLLBACK [WORK] TO [SAVEPOINT] savepoint_name
```

回滚当前事务里的语句所作出的修改，把各有关数据表恢复到修改前的状态。这只适用于支持事务处理的存储引擎。（对于不支持事务处理的存储引擎，每条语句作出的修改在它执行完毕后立刻生效，因而无法回滚。）

可选的关键字 WORK 目前没有任何效果。CHAIN 和 RELEASE 子句的效果与 COMMIT 语句的同名子句相同，详见 COMMIT 语句条目。

如果给出了 TO SAVEPOINT 子句，这条语句将只把当前事务回滚到给定的保存点。这个子句适用于 InnoDB 和 Falcon 事务。

如果没在事先使用 START TRANSACTION 语句或是通过把 autocommit 变量设置为 0 的办法禁用 autocommit 模式，ROLLBACK 语句将什么也不做。

WORK、CHAIN 和 RELEASE 子句都是从 MySQL 5.0.3 版开始引入的，SAVEPOINT 关键字则从那时开始变成了可选的。

- SAVEPOINT

```
SAVEPOINT savepoint_name
```

创建一个名为 *savepoint_name* 的事务还原点，任何与之同名的现有保存点将被删除。在当前事务里，可以用 ROLLBACK TO SAVEPOINT 语句把当前事务回滚到给定的保存点。

● SELECT

```
SELECT
  [select_option] ...
  select_expr [, select_expr] ...
  [FROM tbl_refs
  [WHERE where_expr]
  [GROUP BY {col_name | expr | position} [ASC | DESC], ...  [WITH ROLLUP]]
  [HAVING where_expr]
  [ORDER BY {col_name | expr | position} [ASC | DESC], ...]
  [LIMIT {[skip_count,] show_count | show_count OFFSET skip_count}]
  [PROCEDURE procedure_name([param_list])]
  [
      INTO OUTFILE 'file_name' [field_options] [line_options]
    | INTO DUMPFILE 'file_name'
    | INTO var_name [, var_name] ...
  ]
  [FOR UPDATE | LOCK IN SHARE MODE] ]
```

SELECT 语句通常用来从数据表里检索信息，但因为 SELECT 语句中除 SELECT 关键字和 *select_list* 子句之外的成分都是可选的，所以还可以用它对表达式进行求值，如下所示：

```
SELECT 'one plus one =', 1+1;
```

为了与那些要求 SELECT 语句必须有一个 FROM 子句的数据库系统保持兼容，MySQL 准备了一个 DUAL 关键字来充当“虚假”数据表：

```
SELECT 'one plus one =', 1+1 FROM DUAL;
```

子查询是嵌套在一个 SELECT 语句里的另一个 SELECT 语句，在 2.9 节里可以找到很多例子。子查询还可以用在 DELETE 和 UPDATE 语句的 WHERE 子句里以及 INSERT 和 REPLACE 语句里。不过，MySQL 不允许使用子查询选取你正在修改的数据表。

select_options 子句可以包含一个或者多个下列选项：

❑ ALL、DISTINCT、DISTINCTROW

这几个选项控制着是否需要返回重复的数据行。ALL 表示将返回所有的数据行，它是默认值。DISTINCT 和 DISTINCTROW 表示将从结果集里剔除重复的数据行。

❑ HIGH_PRIORITY

指定 HIGH_PRIORITY 使语句有更高的优先级（如果原本要等待的话）。如果有客户程序正在用 SELECT 语句读取某个数据表，而且 SELECT 语句有 HIGH_PRIORITY 选项，那么对这个数据表进行写操作的其他语句（比如 INSERT 和 UPDATE）就必须等待这个读操作完成后才能执行，这条 SELECT 语句有高于那些写操作语句的优先级。不过，因为这个选项会延缓写语句的执行，所以应该只在那些会很快执行完成并且需要立刻执行的 SELECT 语句里才使用这个选项。HIGH_PRIORITY 选项只适用于使用表级别锁定的存储引擎，如 MyISAM、MEMORY 和 MERGE。

❑ SQL_BUFFER_RESULT

在处理完 SELECT 语句之后，MySQL 还要花上点时间去把查询结果发送回客户端。在此期间，如果这条 SELECT 语句不带 SQL_BUFFER_RESULT 选项，各有关数据表将仍处于锁定状态；而如果这条 SELECT 语句带 SQL_BUFFER_RESULT 选项让服务器把查询结果另外存放到一个临时表里并解除数据表的锁定状态。换句话说，SQL_BUFFER_RESULT 选项将使 MySQL 服务器尽可能早地解除锁定状态，从而使其他客户程序尽可能早地访问那些数据表，但使用这个选项会消耗更多的磁盘空间和内存。

- SQL_CACHE、SQL_NO_CACHE

  如果查询结果是可以缓存的，并且查询缓存处于 DEMAND 模式，SQL_CACHE 选项将导致 SELECT 语句的查询结果被缓存起来。SQL_NO_CACHE 选项将禁用对查询结果的各种缓存机制。
- SQL_CALC_FOUND_ROWS

  在默认的情况下，带有 LIMIT 子句的 SELECT 语句所返回的数据行计数值将等于实际返回的数据行的个数。如果你在这类 SELECT 语句里使用了 SQL_CALC_FOUND_ROWS 选项，MySQL 服务器将把不带 LIMIT 时返回的行数也统计出来。如果想知道这个计数值到底是多少，请在该 SELECT 语句返回后立刻发出一个 SELECT FOUND_ROWS()语句。
- SQL_BIG_RESULT、SQL_SMALL_RESULT

  这两个关键字反映结果集尺寸是小还是大，优化器可以根据这一信息更好地处理 SELECT 语句。
- STRAIGHT_JOIN

  强制数据表必须按它们在 FROM 子句中的先后顺序进行联结。如果你认为优化器作出的不是最佳选择，就可以利用这个选项让 SELECT 查询按你指定的顺序去检索数据表。

select_list 子句用来列举 SELECT 语句将要返回的输出列，多个输出列之间要用逗号彼此分隔开。输出列可以是数据表中的数据列，也可以是 MySQL 表达式（包括标量子查询）。你还可以利用 *AS alias_name* 语法（AS 关键字是可选的）给输出列起一个别名，别名将成为输出报告中的列标题，并可以用在 GROUP BY、ORDER BY 和 HAVING 等子句里。注意：在 WHERE 子句里不允许使用别名。

星号（*）代表“FROM 子句所给定的数据表里的全体数据列”，而 *tbl_name*.*则代表“数据表 *tbl_name* 里的全体数据列”。

FROM 子句用来列出 SELECT 语句将从中选取数据行的数据表。MySQL 支持以下联结语法：

```
tbl_refs:
   tbl_ref[.tbl_ref]...

tbl_ref:
   tbl_factor
  | join_tbl

tbl_factor:
    tbl_name
  | (subquery) [AS] alias_name
  | (tbl_refs)
  | { OJ tbl_ref LEFT OUTER JOIN tbl_ref ON conditional_expr }

join_tbl:
    tbl_ref [INNER | CROSS] JOIN tbl_factor [join_condition]
  | tbl_ref STRAIGHT_JOIN tbl_factor [ON conditional_expr]
  | tbl_ref {LEFT | RIGHT} [OUTER] JOIN tbl_ref join_condition
  | tbl_ref NATURAL [{LEFT | RIGHT} [OUTER]] JOIN tbl_factor

join_condition:
    ON conditional_expr
  | USING (col_list)
```

注意　上面给出的语法从MySQL 5.0.12版开始生效，该版本对以前的语法做了一些修改以求和SQL语言标准更好地兼容。如果你有兴趣了解上述语法与MySQL 5.0.12版之前的相关语法有什么不同，请参阅《MySQL参考手册》。

每个数据表名都可以有一个别名或者索引提示。在 SELECT 语句里引用数据表的完整语法如下所示：

```
tbl_name
  [[AS] alias_name]
  [{USE | IGNORE | FORCE} {INDEX | KEY}
    [FOR {JOIN | ORDER BY | GROUP BY}]
    (index_list)]
```

如果需要在 FROM 子句里给数据表起一个别名，可以使用 *tbl_name alias_name* 或者 *tbl_name* AS *alias_name* 语法。别名机制使我们能够在查询命令中的其他地方利用别名来引用数据表里的数据列。

MySQL 还允许在 FROM 子句里使用子查询的结果集来充当数据表，但必须把子查询用括号括起来，并给它起一个别名以便在 SELECT 语句中的其他地方引用这样的“数据表”，如下所示：

```
SELECT * FROM (SELECT 1) AS t;
```

USE INDEX、IGNORE INDEX 和 FORCE INDEX 子句的作用是为优化器提供必要的索引提示。它们可以帮助优化器正确选择索引来联结数据表。USE INDEX 子句告诉优化器只能从 *index_list* 列表中选择。IGNORE INDEX 子句告诉优化器不要使用哪些索引。FORCE INDEX 子句的含义和 USE INDEX 子句相似，但向优化器特别强调了应该尽最大努力选用索引而避免使用全表扫描。

index_list 是由一个或多个以逗号分隔的索引名构成的一个列表（但有一个例外，我们马上就会讲到），其中列出的每一个索引都必须是来自给定数据表的索引名字或者是代表给定数据表的 PRIMARY KEY 的 PRIMARY 关键字。

在 MySQL 5.0.40 版之前，索引提示只用于选取数据行和联结数据表，不用于处理 ORDER BY 或 GROUP BY 子句。在 5.0.40 版以后的 MySQL 5.0 系列版本里，你可以使用 FOR JOIN 子句来明确表达这种行为。在 5.1.17 版以后的 MySQL 5.1 系列版本里，仍可以那样使用 FOR JOIN 子句，但新增加了几个与索引提示有关的修改。

- 不带 FOR 子句的提示除了可以用于选取数据行和联结数据表（像以前一样），还可以用于处理 ORDER BY 或 GROUP BY 子句。
- USE 子句的 *index_list* 列表可以是空白，其含义是“不使用任何索引”。
- 允许每个 *tabl_name* 带有多个索引提示，但不允许为同一个 *tabl_name* 数据表同时使用 USE INDEX 和 FORCE INDEX 子句。

索引提示对 FULLTEXT 索引没有任何效果。

联结型 SELECT 语句将按照以下描述从给定的数据表选取数据行，但实际返回到客户端的数据行的个数会受限于 WHERE、HAVING 或 LIMIT 子句。

- 如果 FROM 子句只列举了一个数据表，SELECT 将从该数据表检索数据行。
- 如果 FROM 子句以逗号为间隔列举了多个数据表，SELECT 将返回这些数据表里的数据行的全排列组合。如果没有 NO 或 USING 子句，使用 JOIN、CROSS JOIN 或 CROSS JOIN 与使用逗号等价。STRAIGHT_JOIN 与此类似，但强制 MySQL 优化器必须按数据表在 FROM 子句中的先后顺序来。如果你认为 MySQL 优化器做出的不是最佳选择，就可以用这个选项来指导它。
- 与逗号操作符不同，使用 JOIN、CROSS JOIN 或 INNER JOIN 执行的联结操作可以用 ON 或 USING 子句来约束表间的匹配。匹配行是根据 ON*conditional_expr* 或 USING(*col_list*) 子句中的条件来确定的。

  Conditional_expr 是将用在 WHERE 子句中的表达式的形式。*col_list* 由 1 个或多个以逗号分隔的列名组成，每个列名必须是在两个联结的表中都出现的列的名字。

- 在从数据表检索数据行时，LEFT JOIN 将强制性地为左数据表里的每一个数据行生成一个数据行，哪怕右数据表里没有有与之匹配的数据行也是如此。当没有匹配的时候，来自右数据表的数据列将被返回为 NULL 值。

  表名称后面的 ON 或 USING()子句与 JOIN、CROSS JOIN 和 INNER JOIN 里一样。LEFT OUTER JOIN 是 LEFT JOIN 的一个同义词。OJ 语法与此类似，这个语法是为了使 MySQL 与 ODBC 标准保持兼容而引入的。注意：OJ 语法中的花括号不是元字符，它们必须原样出现在相应的 SELECT 语句中。
- NATURAL LEFT JOIN 相当于 LEFT JOIN USING (*column_list*)，这里的 *column_list* 必须把同时出现在两个数据表里的数据列全都列举出来。
- RIGHT JOIN 类型与相应的 LEFT JOIN 类型相似，但前者把数据表的角色。
- 逗号联结的优先级低于其他类型的联结。混合使用逗号关联和其他联结类型有可能导致"Unknown column"（未知数据列）错误，把逗号替换为 INNER JOIN 通常可以解决这样的问题。

MySQL 将根据 WHERE 子句所给出的条件表达式从 FROM 子句所列举的数据表里选取数据行。不满足该条件表达式的数据行将不会出现在 SELECT 语句的查询结果里。结果集还要受到 HAVING 和 LIMIT 子句的进一步限制。注意：数据列别名是不允许用在 WHERE 子句里的。

GROUP BY 和 ORDER BY 子句的语法相似。GROUP BY *col_list* 子句将根据 col_list 所给定的数据列对结果集里的数据行进行归组。当你在 *select_List* 子句里使用了 COUNT()或 MAX()等汇总函数时，就需要使用 GROUP BY 子句。ORDER BY col_list 规定结果集应按指定数据列排序。在这两个子句中，你可以使用数据列的名称、别名或者数据列在 *select_list* 子句里的位置来引用数据列。数据列的位置序号是无符号整数，以 1 开始，但使用数据列的位置是非标准的，已经弃用。你还可以用表达式来分组，或根据表达式结果排序。例如，ORDER BY RAND()是以随机顺序为数据行排序的。

在 GROUP BY 和 ORDER BY 子句里，可以在任何一个数据列的名字后面加上 ASC 或 DESC 关键字以表明该数据列应该按照递增或递减的顺序来排序。如果没有这两个关键字中的任何一个，数据列将默认使用递增顺序。这两个关键字还可以用在 GROUP BY 子句里，这是因为 MySQL 中的 GROUP BY 子句不仅会对数据行进行分组，还会对分组结果进行排序。如果同时给出了 GROUP BY 和 ORDER BY 子句，检索结果的排序顺序将由 ORDER BY 子句决定。你可以加上一条 ORDRE BY NULL 子句来禁用 GROUP BY 子句的这种隐式排序效果（这将减少不必要的排序开销）。

在 GROUP BY 子句的末尾还可以给出 WITH ROLLUP 限定符，它将在每个数据行分组的后面加上一个关于该分组的"小计"数据行，还将在输出报告的末尾加上一个"总计"数据行。

MySQL 将根据 HAVING 子句所给出的次要条件表达式对那些已经满足 WHERE 子句所给出的主要条件表达式（并且已根据 GROUP BY 子句分组）的数据行做进一步的筛选。不满足 HAVING 条件的数据行将不会出现在 SELECT 语句的查询结果里。HAVING 子句非常适合因带有汇总函数而无法用在 WHERE 子句里的条件表达式。但是，如果某个条件表达式在 WHERE 子句和 HAVING 子句都是合法的，就应该把它放到 WHERE 子句里，因为只有 WHERE 子句里的条件表达式才会得到优化器的分析和优化。

LIMIT 子句的用途是从结果集里进一步选取它的某个组成部分。这个子句可以带一个或者两个参数（这些参数必须是整数常数）。LIMIT *n* 将返回结果集里的前 *n* 个数据行，LIMIT *m*, *n* 则跳过结果集里的前 *m* 个数据行，返回随后的 *n* 个数据行。

PROCEDURE 子句给出的是一个代码模块的名字，服务器会在把结果集发送回客户程序之前先发送到这个代码模块去进行一些处理。参数表 *param_list* 可以为空，也可以是一个用逗号分隔的参数清

单，这些参数将被传递到指定的代码模块里去。你可以通过 PROCEDURE ANALYSE()子句来获得该 SELECT 语句所选取的数据列里的数据值的信息。

各种 INTO 格式都指定了查询结果可选的目的地。如果使用了 USE 子句，语句就不能用作内嵌的 SELECT。INTO 跟在 select_expr 之后，提早指定可选的目的地。

SELECT 语句的结果可以使用 INTO OUTFILE'file_name'子句写入名为 *file_name* 的文件中。*file_options* 和 *line_options* 子句的语法与 LOAD DATA 语句的相应子句一样，参见 LOAD DATA 条目。

INTO DUMPFILE'*file_nam*'与 INTO OUTFILE 类似，但它只写一行，并且写出整个输出，不加以解释。也就是，它写出初始值，不用分隔符、引号或结束符。如果你想编写 BLOB 数据到文件中，如一个图像或其他二进制数据，那么它很有用。

INTO OUTFILE 和 INTO DUMPFILE 选项用来确定输出文件位置，所用规则与不带 LOCAL 选项的 LOAD DATA 语句的规则一样。你必须具备 FILE 权限，输出文件必须尚不存在——服务器会在服务器主机上把它创建为一个允许全局访问的文件，并把其属主设置为运行服务器的那个账户。从 MySQL 5.0.19/5.1.6 版开始，对文件名的解析将使用 character_set_filesystem 系统变量所指定的那个字符集来进行。

如果 INTO 关键字的后面是一个以逗号分隔的变量名列表，SELECT 语句将把检索结果存入那些变量。这些变量既可以是一个@*var_name* 形式的用户定义变量，也可以是包含这条 SELECT 语句的存储例程的某个参数或局部变量。这种查询必须选取且只能选取一个数据行，而每个输出列都必须有一个与之对应的变量。

FOR UPDATE 和 LOCK IN SHARE MODE 子句将锁定 SELECT 语句所选取的数据行，直到当前事务被提交或者回滚。这在多语句事务里非常有用。在支持数据行级锁定的数据表（如 InnoDB）上使用 FOR UPDATE 子句将在选取的数据行上施加一个独占性的写操作锁。使用 LOCK IN SHARE MODE 子句将在数据行上施加一个读操作锁，即允许其他客户程序去读这些数据行、但不允许修改这些数据行。注意，如果查询优化器发现没有用于检查数据行的索引，就必须扫描（并锁定）表中的所有行。

下面这些语句演示了 SELECT 语句的一些用法。本书的第 1 章和第 2 章里还有更多的示例。

- 选取某数据表的全部内容：

```
SELECT * FROM president;
```

- 选取全部内容，但要根据总统姓名排序：

```
SELECT * FROM president ORDER BY last_name, first_name;
```

- 把出生日期是或者晚于'1900-01-01'的总统查出来：

```
SELECT * FROM president WHERE birth >= '1900-01-01';
```

- 把出生日期是或者晚于'1900-01-01'的总统查出来，但要按出生日期排序：

```
SELECT * FROM president WHERE birth >= '1900-01-01' ORDER BY birth;
```

- 查看 member 数据表里的数据行涉及哪些州：

```
SELECT DISTINCT state FROM member;
```

- 选取 member 数据表里的数据行并把它们写到一个文件里，各数据列以逗号分隔：

```
SELECT * INTO OUTFILE '/tmp/member.txt'
  FIELDS TERMINATED BY ',' FROM member;
```

- 把某次考试前 5 名学生的记录选取出来：

```
SELECT * FROM score WHERE event_id = 9 ORDER BY score DESC LIMIT 5;
```

● SET

```
SET [OPTION] assignment [, assignment ] ...

assignment: var_name = expr
```

SET 语句用来对系统变量、用户定义变量或存储程序本地变量赋值。附录 D 提供了有关系统和用户定义变量的信息，E.2.2 节描述了存储程序本地变量的声明语法。SET 也用于一些其他设置，后面将会介绍。

其他以 SET 开关的语句（SET PASSWORD 和 SET TRANSACTION）将在后面单独介绍。

当 SET 用于为变量赋值时，每个赋值语句中的 *var_name* 就是待赋值的变量，*expr* 表达式指定要赋给变量的值。赋值操作符可以是“=”或者是“:=”。

SET 语句可以用来对用户定义的变量进行赋值，如下所示：

```
SET @day = CURDATE(), @time = CURTIME();
```

SET 还可以给系统变量赋值，其中许多是动态的，你在服务器仍在运行时可以修改它们。MySQL 提供的动态系统变量有两类：全局级变量的有效范围是整个服务器，它们对所有的客户程序都有影响；会话级变量（也叫做本地变量）的有效范围只局限于某个特定的客户连接。对于那些同时存在于系统、会话两个级别的变量，每个新建的客户连接都会根据相应的全局级变量的值得到一个初始的会话级变量设置。任何一个客户程序都可以修改它自己的会话级变量，但只有那些具备 SUPER 权限的用户才能修改全局级变量。

设置系统变量的语法有好几种。全局级变量可以用下面两条语句中的任何一条去修改（以 sql_mode 为例）：

```
SET GLOBAL sql_mode = 'ANSI_QUOTES';
SET @@GLOBAL.sql_mode = 'ANSI_QUOTES';
```

会话级变量则需要用 SESSION 代替 GLOBAL：

```
SET SESSION sql_mode = 'ANSI_QUOTES';
SET @@SESSION.sql_mode = 'ANSI_QUOTES';
```

还可以把 LOCAL 用作 SESSION 的一个同义词，如下所示：

```
SET LOCAL sql_mode = 'ANSI_QUOTES';
SET @@LOCAL.sql_mode = 'ANSI_QUOTES';
```

如果在SET语句里没有给出GLOBAL、SESSION或LOCAL关键字，SET语句将默认修改会话级变量：

```
SET sql_mode = 'ANSI_QUOTES';
SET @@sql_mode = 'ANSI_QUOTES';
```

要了解系统变量的值，请使用 SHOW VARIABLES 语句。你可以使用 SELECT 检索各个系统变量的值。

```
SELECT @@GLOBAL.sql_mode, @@SESSION.sql_mode, @@LOCAL.sql_mode;
```

12.6.1 节对系统变量的用途和用法进行了详细的讨论。

下面是可以通过 SET 语句加以控制的其他一些选项：

❑ SET CHARACTER SET {*charset* | DEFAULT}

把 character_set_client 和 character_set_results 会话变量设置为给定的字符集，把 character_set_connection 会话变量设置为 character_set_database 变量的值。这些变量影响着发送至服务器和从其发出的字符数据的转换。ucs2、utf18 或 utf32 目前还不是有

效的 *charset* 值。

SET CHARACTER SET DEFAULT 可以恢复字符集的默认设置。

- SET NAMES {*charset* | '*charset*' | DEFAULT}

  把 character_set_client、character_set_connection 和 character_set_results 会话变量设置为给定的字符集，把 collation_connection 设置为 character_set_connection 的默认排序方式。这些变量影响着进入和离开服务器的字符数据的转换。ucs2、utf18 或 utf32 目前还不是有效的 *charset* 值。

  SET NAMES DEFAULT 语句将恢复各有关字符集的默认设置。

- SET PASSWORD

```
SET PASSWORD [FOR account] = PASSWORD('pass_val')
SET PASSWORD [FOR account] = OLD_PASSWORD('pass_val')
SET PASSWORD [FOR account] = 'encrypted_pass_val'
```

SET PASSWORD 语句用来修改某个 MySQL 账户的口令。你可以修改自己的口令，但作为一个匿名用户连接到服务器时例外。如果想修改另一个账户的口令，必须具备 mysql 数据库的 UPDATE 权限。

如果不带 FOR 子句，这条语句将修改当前账户的口令。如果带有 FOR 子句，它将修改指定账户的口令，账户名必须按照'*user_name*'@'*host_name*'的格式给出，详见 12.4.1 节的第 1 小节。

口令值'*pass_val*'必须用 PASSWORD()函数进行标准化加密或使用 OLD_PASSWORD()函数进行老式（MySQL 4.1 版之前）加密。如果没有使用这两个函数中的任何一个，相应的'*encrrypted_pass_val*'值必须是一个已经加过密的口令字符串。

```
SET PASSWORD = PASSWORD('secret');
SET PASSWORD FOR 'paul' = PASSWORD('secret');
SET PASSWORD FOR 'paul'@'localhost' = PASSWORD('secret');
SET PASSWORD FOR 'bill'@'%.bigcorp.com' = PASSWORD('old-sneep');
```

- SET TRANSACTION

```
SET [GLOBAL | SESSION] TRANSACTION ISOLATION LEVEL level
```

这条语句用来设置事务处理的隔离级别。

- 如果给出了 GLOBAL 选项，它将设置全局级（适用于服务器全局）隔离级别，此后连接到服务器的所有客户都将以此作为默认的隔离级别。
- 如果给出了 SESSION 选项，它将设置会话级（只适用于当前客户）隔离级别，当前会话里此后发生的事务都将以此作为其默认的隔离级别。
- 如果这两个选项都没有给出，它将只设置当前会话里的下一个事务的隔离级别。

必须具备 SUPER 权限才能设置全局级隔离级别。任何一个客户都可以改变自己的会话级和下一个事务的隔离级别。

level 部分所代表的隔离级别必须是下列值之一。

- READ UNCOMMITTED

  某给定事务可以看到其他事务对数据行作的修改，不管它们是否已经提交。

- READ COMMITTED

  某给定事务只能看到其他事务已经提交了的数据行改动。

- REPEATABLE READ

  如果在某个事务里两次执行同一条 SELECT 语句，其结果是可重复的。换句话说，无论是否有

其他事务在此期间修改或插入了数据行，同一个事务里的同一个 SELECT 语句每次都能得到同样的结果。

- SERIALIZABLE

  这个隔离级别类似于 REPEATABLE READ，但对事务的隔离更加彻底：在某个事务结束之前，发生在它之后的其他事务不能对它正在选取的数据行修改。InnoDB 存储引擎支持这个隔离级别，Falcon 存储引擎目前还不支持它。

SET TRANSACTION 语句适用于 InnoDB 和 Falcon 存储引擎。默认的隔离级别是 REPEATABLE READ。不支持事务处理的存储引擎没有隔离级别的概念。

2.13.3 节对事务的隔离问题以及各种隔离级别做了详细的讨论。

- SHOW

```
SHOW BINLOG EVENTS
SHOW CHARACTER SET
SHOW COLLATION
SHOW COLUMNS
SHOW CREATE DATABASE
SHOW CREATE EVENT
SHOW CREATE {FUNCTION | PROCEDURE}
SHOW CREATE TABLE
SHOW CREATE TRIGGER
SHOW CREATE VIEW
SHOW DATABASES
SHOW ENGINE
SHOW ENGINES
SHOW ERRORS
SHOW EVENTS
SHOW {FUNCTION | PROCEDURE} STATUS
SHOW GRANTS
SHOW INDEX
SHOW INNODB STATUS
SHOW {MASTER | BINARY} LOGS
SHOW MASTER STATUS
SHOW MUTEX STATUS
SHOW OPEN TABLES
SHOW PRIVILEGES
SHOW PROCESSLIST
SHOW SLAVE HOSTS
SHOW SLAVE STATUS
SHOW STATUS
SHOW TABLE STATUS
SHOW TABLE TYPES
SHOW TABLES
SHOW TRIGGERS
SHOW VARIABLES
SHOW WARNINGS
```

SHOW 语句的各种形式使你能够查知关于数据库、数据表、数据列、存储程序以及服务器运行状态的各种信息。有几种 SHOW 语句允许使用一个可选的 FROM *db_name* 子句，这个子句的作用是告诉 MySQL 你想了解的是关于哪一个数据库的信息；如果你没有给出这个子句，MySQL 就会把关于当前默认数据库的信息报告给你。有些语句支持使用 FROM 指定数据表或数据库名称，在这些语句中，IN 可以作为同义词使用。

此外，有几种 SHOW 语句允许使用一个可选的 LIKE '*pattern*'子句，这个子句的作用是让 SHOW 语句只显示那些与给定模式相匹配的值。'*pattern*'将被解释为一个 SQL 匹配模式且允许包含“%”或“_”通配符。

从 MySQL 5.0.2 版开始，新增的 INFORMATION_SCHEMA 数据库为我们提供了另外一种获得数据库元数据的方法，许多通过 SHOW 语句获得的信息现在还可以从 INFORMATION_SCHEMA 数据库里的数据表查到。在此基础上，那些支持 LIKE '*pattern*'子句的 SHOW 语句现在还可以改用一条 WHERE 子句对它将显示的信息进行筛选。如果你有兴趣了解更多这方面的信息，请参阅 2.7 节。

- SHOW BINLOG EVENTS

```
SHOW BINLOG EVENTS [IN 'file_name'] [FROM position]
[LIMIT [skip_count,] show_count]
```

在复制机制中的主服务器上发出这条语句将显示二进制日志里的事件，事件与 SQL 语句大致对应。它的输出报告由以下几个输出列组成。

- ❑ Log_name
  二进制日志文件的名字。
- ❑ Pos
  事件在日志文件中的位置。
- ❑ Event_type
  事件的类型，比如说，可执行 SQL 语句的类型是 Query。
- ❑ Server_id
  负责记录这个事件的服务器的 ID 编号。
- ❑ End_log_pos
  记载本次事件后，日志文件里的下一个字节的位置。
- ❑ Info
  事件信息，比如 Query 事件的语句文本。

SHOW BINLOG EVENT 需要你具备 REPLICATION SLAVE 权限才能执行。

- SHOW CHARACTER SET

```
SHOW CHARACTER SET [LIKE 'pattern' | WHERE where_expr]
```

显示服务器当前支持的字符集。如果包含 LIKE 子句，则只显示其名称与给定模式相匹配的字符集的信息。如果包含一条 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

SHOW CHARACTER SET 语句的输出包含以下输出列：

- ❑ Charset
  简写的字符集名字。这些简写的名字可以直接用在 SQL 语句里。
- ❑ Description
  一个描述性的字符集名字。
- ❑ Default collation
  该字符集的默认排序方式的名字。
- ❑ Maxlen
  该字符集里最“宽”的字符的长度，以字节单位。对于多字节字符集，这个值将大于 1。对于单字节字符集，所有字符都只占用一个字节，所以这个值将是 1。

- SHOW COLLATION

```
SHOW COLLATION [LIKE 'pattern' | WHERE where_expr]
```

显示每一种字符集的所有可用排序方式。如果包含一条 LIKE 子句，则只显示其名称与给定模式

相匹配的排序方式的信息。如果包含一条 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

SHOW COLLATION 语句的输出包含以下输出列。

- ❑ Collation
  排序方式的名字。
- ❑ Charset
  与这种排序方式相关联的字符集的名字。
- ❑ Id
  排序方式的 ID 编号。
- ❑ Default
  如果这种排序方式是相关字符集的默认排序方式，这里将是 Yes；否则，这里将是空白。
- ❑ Compiled
  如果这种排序方式已被编译到服务器里，这里将是 Yes；否则，这里将是空白。
- ❑ Sortlen
  一个与内存耗用量有关的开销因数，在按照这种排序方式对数据值排序时，服务器将根据这个因数为其内部的字符串转换操作分配内存。

- SHOW COLUMNS

```
SHOW [FULL] COLUMNS {FROM | IN} tbl_name
  [{FROM | IN} db_name] [LIKE 'pattern' | WHERE where_expr]
```

显示给定数据表（或视图，从 MySQL 5.0.1 版开始）的数据列。这条语句的输出只包含那些你拥有权限的数据列。SHOW FIELDS 语句是 SHOW COLUMNS 语句的一个同义词。如果包含 FULL 关键字，这条语句将增加显示 Collation、Privilege 和 Comment 输出字段。如果包含一条 LIKE 子句，则只显示其名称与给定模式相匹配的数据列的信息。如果包含一条 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

要指定包含给定数据表的数据库的名字，可以使用一条 FROM *db_name* 子句或以 *db_name.tbl_name* 格式写出数据表名：

```
SHOW COLUMNS FROM president;
SHOW COLUMNS FROM president FROM sampdb;
SHOW COLUMNS FROM sampdb.president;
```

SHOW COLUMNS 语句的输出可以提供数据表里的每一列的以下类型的信息：

- ❑ Field
  该数据列的名字。
- ❑ Type
  该数据列的类型。在类型名的后面可能还会列出一些相关的属性。
- ❑ Collation
  非二进制字符串数据列的排序方式的名字、如果是其他类型的数据列，这里将是 NULL。排序方式的名字里隐含着字符集的名字。这个信息只在你给出了 FULL 关键字时才会出现。
- ❑ Null
  Yes 表示该数据列允许包含 NULL 值，但在 MySQL5.0.3 之前为空，从 MySQL5.0.3 起为 NO。
- ❑ Key
  是否有建立在该数据列上的索引。
- ❑ Default

该数据列的默认值。

- ❑ Extra
  提取与该数据列有关的信息。如果数据列有 AUTO_INCREMENT 属性，则显示 auto_increment，否则为空。
- ❑ Privileges
  你在该数据列上的操作权限，这部分信息只有在你给出了 FULL 关键字时才会显示出来。
- ❑ Comment
  定义该数据列时在 COMMENT 子句里给出的注释信息。这部分信息只有在你给出了 FULL 关键字时才会显示出来。

- SHOW CREATE

```
SHOW CREATE DATABASE [IF NOT EXISTS] db_name
SHOW CREATE EVENT event_name
SHOW CREATE FUNCTION func_name
SHOW CREATE PROCEDURE proc_name
SHOW CREATE TABLE tbl_name
SHOW CREATE TRIGGER trigger_name
SHOW CREATE VIEW view_name
```

SHOW CREATE *obj_type* 语句将显示创建给定对象的 CREATE *obj_type* 语句。这条语句的几种形式还可以显示一些关于给定对象的其他信息，例如当初创建该对象时的 sql_mode 值。

对 SHOW CREATE DATABASE 语句而言，如果它包含 IF NOT EXIST 子句，它输出的 CREATE TABLE 语句也将包含该子句。

SHOW CREATE VIEW 语句是从 MySQL 5.0.1 版开始引入的，SHOW CREATE EVENT 语句从 MySQL 5.1.6 版开始，SHOW CREATE TRIGGER 语句则从 MySQL 5.1.21 版开始。

- SHOW DATABASES

```
SHOW DATABASES [LIKE 'pattern' | WHERE where_expr]
```

显示 MySQL 服务器主机上当前可以使用的数据库。如果包含 LIKE 子句，则只显示其名称与给定模式相匹配的数据库的信息。如果包含 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

如果没有 SHOW DATABASES 权限，你将只能看到你有某种访问权限的数据库。如果服务器是用 --skip-show-database 选项启动的，并且你具备 SHOW DATABASES 权限，你将看到所有的数据库；否则，一个数据库也看不到。

- SHOW ENGINE

```
SHOW ENGINE engine_name info_type
```

这条语句将显示关于给定存储引擎的信息。对于 InnoDB 存储引擎，这条语句还有以下几种形式可供选用。

- ❑ SHOW ENGINE INNODB STATUS
  显示关于 InnoDB 存储引擎内部操作状态的信息。这条语句是 SHOW INNODB STATUS 语句的升级替代版。它要求你必须具备 PROCESS 权限（MySQL 5.1.24 版之前是 SUPER 权限）。
- ❑ SHOW ENGINE INNODB MUTEX
  显示关于 InnoDB 互斥锁定机制的信息。这条语句是从 MySQL 5.1.2 版开始引入的，它是 SHOW MUTEX STATUS 语句的升级替代版。它要求你必须具备 PROCESS 权限（MySQL 5.1.24 版之前

是 SUPER 权限）。

- SHOW ENGINES

```
SHOW [STORAGE] ENGINES
```

显示服务器当前支持使用的存储引擎。这条语句将告诉你 MySQL 对每种存储引擎的支持级别，并提供关于存储引擎特性的简要描述。

这条语句的输出报告包含以下几个输出列。

❑ Engine
存储引擎的名字（MyISAM、InnoDB 等）

❑ Support
对给定存储引擎的支持级别：YES 是支持；NO 是不支持；DISABLED 是支持但在运行时被禁用；DEFAULT 则表示该存储引擎是默认的存储引擎，默认的存储引擎总是可用的。

❑ Comment
关于给定存储引擎的描述性文字。

❑ Transactions
该引擎是否支持事务处理。

❑ XA
该引擎是否支持分布式事务处理。

❑ Savepoints
该引擎是否支持半程事务回滚。

Transactions、XA 和 Savepoints 输出列是从 MySQL 5.1.2 版开始引入的。

- SHOW ERRORS

```
SHOW ERRORS [LIMIT [skip_count,] show_count]

SHOW COUNT(*) ERRORS
```

SHOW ERRORS 语句类似于 SHOW WARNINGS 语句，但只显示问题足够严重的出错消息。SHOW COUNT(*) ERRORS 语句类似于 SHOW COUNT(*) WARNINGS 语句，但显示的是 error_count 变量的值而不是 warning_count 变量的值。更多信息请参阅 SHOW WARNINGS 语句条目。

- SHOW EVENTS

```
SHOW EVENTS [FROM db_name] [LIKE 'pattern' | WHERE where_expr]
```

这条语句将显示关于默认数据库里的事件的信息，如果带有 FROM 子句，则显示关于给定数据库里的事件的信息。如果包含 LIKE 子句，则只显示其名称与给定模式相匹配的事件的信息。如果包含 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

SHOW EVENTS 语句是从 MySQL 5.1.6 版开始引入的。

- SHOW FUNCTION STATUS、SHOW PROCEDURE STATUS

```
SHOW{FUNCTION|PROCEDURE}STATUS
  [LIKE'pattern'|WHERE where_expr]
```

这些语句将显示关于默认数据库里的存储函数或存储过程的描述性信息。如果包含 LIKE 子句，则只显示其名称与给定模式相匹配的存储例程的信息。如果包含 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

● SHOW GRANTS

```
SHOW GRANTS [FOR account]
```

这条语句将显示给定用户的访问权限，用户名必须以'*user_name*'@'*host_name*'的格式给出，具体描述参见 12.4.1 节中的第 1 小节。

```
SHOW GRANTS FOR 'root'@'localhost';
SHOW GRANTS FOR ''@'cobra.snake.net';
```

你还可以使用下列语句中的任何一个去查看当前使用的 MySQL 用户账户都有哪些权限：

```
SHOW GRANTS FOR CURRENT_USER();
SHOW GRANTS FOR CURRENT_USER;
SHOW GRANTS;
```

从 MySQL 5.0.24/5.1.12 版开始，当你使用 SHOW GRANTS 语句去查看当前用户账户的权限时，如果该存储过程是在 SQL SECURITY DEFINER 上下文里执行的，将输出关于该存储过程的定义者的信息而不是关于其调用者的。

● SHOW INDEX

```
SHOW {INDEX | KEY} {FROM | IN} tbl_name [{FROM | IN} db_name]
```

显示关于某给定数据表的索引的信息。要指定包含给定数据表的数据库的名字，可以使用一条 FROM *db_name* 子句或是以 *db_name.tbl_name* 的格式写出数据表名：

```
SHOW INDEX FROM score;
SHOW INDEX FROM score FROM sampdb;
SHOW INDEX FROM sampdb.score;
```

SHOW INDEX 语句的输出包含以下输出列：

- Table
  包含该索引的数据表的名字。
- Non_unique
  1 表示该索引允许包含重复的值，0 表示不允许。
- Key_name
  索引的名字。
- Seq_in_index
  数据列在索引中的序号，从 1 开始。
- Column_name
  应用当前输出行的索引中数据列的名字。
- Collation
  数据列在索引中的顺序，它的可取值是 A（升序）、D（降序）或 NULL（不排序）。MySQL 目前还不支持按降序排序的键。
- Cardinality
  索引中独一无二的键值的大致个数。以--analyze 选项启动执行的 myisamchk 将刷新 MyISAM 的这个值，ANALYZE TABLE 语句将刷新 MyISAM 和 InnoDB 数据表的这个值，OPTIMIZE 将刷新 MyISAM 的这个值。
- Sub_part
  如果只对数据列的前 *n* 个字节进行索引，这个输出列将给出该前缀以字节计数的长度。如果是对整个数据列进行索引，这个输出列里的值将是 NULL。

❑ Packed

键的压缩方式，NULL 表示没有压缩。

❑ Null

YES 表示该数据列允许包含 NULL 值，空白则表示不允许。

❑ Index_type

数据列的索引方式，比如 BTREE、FULLTEXT 或 HASH 等。

❑ Comment

保留供该索引的内部注释之用。

● SHOW INNODB STATUS

```
SHOW INNODB STATUS
```

显示关于 InnoDB 存储引擎内部操作状态的信息。它要求你必须具备 SUPER 权限。这条语句目前已被淘汰，替代它的是 SHOW ENGINE INNODB STATUS 语句。

● SHOW MASTER LOGS

```
SHOW {MASTER | BINARY} LOGS
```

这条语句要在复制机制中的主服务器上使用。它将把主服务器上当前可用的二进制日志的名字显示出来。在用 SHOW SLAVE STATUS 语句查明从服务器与二进制日志当前的同步位置之后，发出 PURGE MASTER LOGS 语句之前，通常需要用 SHOW MASTER LOGS 语句去查一下主服务器上的日志文件使用情况。

● SHOW MASTER STATUS

```
SHOW MASTER STATUS
```

这条语句要在主服务器上使用。它能让你了解主服务器上二进制日志文件的状态信息。

SHOW MASTER STATUS 语句的输出报告由以下输出列组成。

❑ File

二进制日志的文件名。

❑ Position

MySQL 服务器在该文件里的当前写位置。

❑ Binlog_Do_DB

一份以逗号分隔的、通过--binlog_do_db 选项明确地与该二进制日志建立复制关系的数据库名单，如果为空，则表示没有这样的数据库。

❑ Binlog_Ignore_DB

一份以逗号分隔的、通过--binlog_ignore_db 选项明确地与该二进制日志解除复制关系的数据库名单，如果为空，则表示没有这样的数据库。

● SHOW MUTEX STATUS

```
SHOW MUTEX STATUS
```

这条语句将显示关于 InnoDB 互斥锁定机制的信息。它是从 MySQL 5.0.3 版开始引入的，在 MySQL 5.1 系列版本里被重新命名为 SHOW ENGINE INNODB MUTEX。

● SHOW OPEN TABLES

```
SHOW OPEN TABLES [{FROM | IN} db_name]
```

```
[LIKE 'pattern' | WHERE where_expr]
```

这条语句将显示一份已在数据表缓存里注册并处于打开状态的非 TEMPORARY 数据表的清单，该清单只包含你在其上拥有某种权限的那些数据表。如果包含 LIKE 子句，则只显示其名称与给定模式相匹配的数据表的信息。如果包含 WHERE 子句，则只输出那些能够满足给定表达式的数据行。这些子句是从 MySQL 5.0.12 版开始引入的。

SHOW OPEN TABLES 语句的输出包含以下输出列。

- ❑ Database
  包含给定数据表的数据库的名字。
- ❑ Table
  数据表的名字。
- ❑ In_use
  数据表目前被使用了多少次。
- ❑ Name_locked
  这个数据表当前是否有一个名字锁，名字锁是在不访问其内容的情况下使用某个数据表（比如说，执行 RENAME TABLE 语句）的必要条件。

- SHOW PRIVILEGES

```
SHOW PRIVILEGES
```

这条语句将显示可以由你授权的权限以及它们的含义。

这条语句的输出报告由以下输出列组成。

- ❑ Privilege
  权限的名字。
- ❑ Context
  该权限的有效范围，比如 Server Admin（系统管理员）、Databases 或 Tables 等。
- ❑ Comment
  关于该权限用途的描述信息。

- SHOW PROCESSLIST

```
SHOW [FULL] PROCESSLIST
```

显示关于当前正在执行的服务器活动的信息。如果你具备 PROCESS 权限，这条语句将显示所有的信息；否则，它将只显示与你本人有关的服务器活动的信息。

- ❑ Id
  客户程序的过程 ID。
- ❑ User
  与该过程相关联的客户的账户名。
- ❑ Host
  该客户是从哪台主机建立的连接。
- ❑ db
  该过程的默认数据库。
- ❑ Command
  正在执行的命令的类型。

- ❑ Time
  过程正在执行的语句所花费的时间，以秒为计量单位。
- ❑ State
  这个输出列能告诉我们 MySQL 对 SQL 语句的处理进行到哪一个阶段了。如果你想反映你在使用 MySQL 时遇到的问题，或者想通过 MySQL 邮件表向别人请教为什么你的线程总停留在某个阶段，就需要用到这个输出列里的信息。
- ❑ Info
  正被执行的查询。使用 FULL 选项将使你查看到数据库查询命令的完整文本，如果没有这个选项，你就只能看到前 100 个字符。

● SHOW SLAVE HOSTS

```
SHOW SLAVE HOSTS
```

这条语句应该在复制机制中的主服务器上使用，它将显示关于当前已在主服务器上注册的从服务器的信息。只有使用--report-host 选项启动的从服务器才会在主服务器上注册。即便是一个已经注册的从服务器，它在 SHOW SLAVE HOSTS 语句的输出报告里会不会有某些特定的输出列还要取决于其他条件。如果没有在启动从服务器的时候使用--report-port 选项，相应的 Port 输出列的值将是空白。如果既没有在启动主服务器的时候使用--show-slave-auth-info 选项、也没有在启动从服务器的时候使用--report-user 和--report-password 选项，相应的 User 和 Password 输出列的值将是空白。

- ❑ Server_id
  从服务器的 ID 号。
- ❑ Host
  从服务器所在的主机名。
- ❑ User
  从服务器用来建立当前连接的账户。
- ❑ Password
  从服务器用来建立当前连接的口令。
- ❑ Port
  从服务器连接在哪一个端口上。
- ❑ Rpl_recovery_tank
  复制恢复级别。
- ❑ Master_id
  主服务器的 ID 号。

● SHOW SLAVE STATUS

```
SHOW SLAVE STATUS
```

这条语句应该在从服务器上使用，它将显示从服务器的复制工作状态信息。SHOW SLAVE STATUS 语句的输出包含以下输出列。

- ❑ Slave_IO_State
  从服务器的 I/O 线程的状态，这里显示的值和 SHOW PROCESSLIST 语句为该线程显示的值是一样的。

- Master_Host
  主服务器的主机名或 IP 地址。
- Master_User
  用来连接主服务器的用户名。
- Master_Port
  用来连接主服务器的端口号。
- Connect_Retry
  等待连接主服务器的秒数。
- Master_Log_File
  主服务器当前使用的二进制日志的文件名。
- Read_Master_Log_Pos
  从服务器的 I/O 线程在主服务器的二进制日志里的当前读位置。
- Relay_Log_File
  当前中继日志文件的名字。
- Relay_Log_Pos
  从服务器在当前中继日志中的读位置。
- Relay_Master_Log_File
  SQL 线程最近执行的事件所在的主服务器上的二进制日志文件的名字。
- Slave_IO_Running
  从服务器的 I/O 线程是否正在运行。
- Slave_SQL_Running
  从服务器的 SQL 线程是否正在运行。
- Replicate_DO_DB
  一份以逗号分隔的、通过--replicate_do_db 选项明确地表明需要进行复制处理的数据库名单，如果为空，则表示没有这样的数据库。
- Replicate_Ignore_DB
  一份以逗号分隔的、通过--replicate_ignore_db 选项明确地表明不需要进行复制处理的数据库名单，如果为空，则表示没有这样的数据库。
- Replicate_Do_Table
  一个以逗号为分隔符、通过--replicate-do-table 选项明确表明需要进行复制处理的数据表名单，如果没有给出任何这样的选项，这个输出列将是空白。
- Replicate_Ignore_Table
  一个以逗号为分隔符、通过--replicate-ignore-table 选项明确表明不需要进行复制处理的数据表名单，如果没有给出任何这样的选项，这个输出列将是空白。
- Replicate_Wild_Do_Table
  一个以逗号为分隔符、通过--replicate-wild_do-table 选项明确表明需要进行复制处理的数据表名单，如果没有给出任何这样的选项，这个输出列将是空白。
- Replicate_ Wild_Ignore_Table
  一个以逗号为分隔符、通过--replicate-wild-ignore-table 选项明确表明不需要进行复制处理的数据表名单，如果没有给出任何这样的选项，这个输出列将是空白。

- Last_Errno

  在 MySQL 5.1.20 之前，这表示最近一次的出错代码。自 MySQL 5.1.20 后，这一列是 Last_SQL_Errno 的别名，0 表示没有出错。
- Last_error

  在 MySQL 5.1.20 之前，这表示最近一次的出错信息，自 MySQL 5.1.20 后，这一列是 Last_SQL_Error 的别名，空白表示没有出错。服务器也会将非空值写入错误日志中。
- Skip_Counter

  从服务器需要跳过（通过设置全局系统变量 sql_slave_skip_counter）的、来自主服务器的日志事件的个数。
- Exec_Master_Log_Pos

  从服务器的 SQL 线程在主服务器的二进制日志里的当前执行位置。
- Relay_Log_Space

  中继日志文件的总长度。
- Until_Condition

  START SLAVE 语句的 UNTIL 子句所给出的条件，该条件决定着 SQL 线程应该在何时停止读取和执行事件。
  - None：START SLAVE 语句没有给出任何 UNTIL 子句。
  - Master：从服务器将一直执行到它的 SQL 线程到达主服务器二进制日志里的给定位置才停止。
  - Relay：从服务器将一直执行到它的 SQL 线程到达中继日志里的给定位置才停止。

如果 Until_Condition 输出列的值是 Master 或 Relay，SQL 线程将在到达 Until_Log_File 和 Until_Log_Pos 输出列的值所指定的文件名和位置时停止执行。

- Until_Log_File

  请参阅 Until_Condition 选项条目里的描述。
- Until_Log_Pos

  请参阅 Until_Condition 选项条目里的描述。
- Master_SSL_Allowed

  是否可以使用 SSL 连接主服务器：Yes 表示可以使用 SSL 连接，No 表示不能使用 SSL 连接，Ignored 表示允许使用 SSL 连接但从服务器在编译时没有启用 SSL 支持。
- Master_SSL_CA_File

  与主服务器建立 SSL 连接时需要提供的 CA（Certificate Authority，证书颁发）文件的路径名，如果没有设定，这个输出列将是空白。
- Master_SSL_CA_Path

  与主服务器建立 SSL 连接时需要提供的 CA 文件所在子目录的路径名，如果没有设定，这个输出列将是空白。
- Master_SSL_Cert

  与主服务器建立 SSL 连接时需要提供的证书文件的路径名，如果没有设定，这个输出列将是空白。
- Master_SSL_Cipher

  以字符串形式给出通过 SSL 连接与主服务器进行加密通信时使用的 SSL 密码，如果没有设定，这个输出列将是空白。

- ❑ Master_SSL_Key
  通过 SSL 连接与主服务器通信时使用的密钥文件的路径名，如果没有设定，这个输出列将是空白。
- ❑ Seconds_Behind_Master
  从服务器落后于主服务器多少秒，其计算方法是用从服务器的 SQL 线程最近一次执行的主复制事件里记录的时间戳减去当前时间。如果 SQL 线程已经赶上了 I/O 线程并处于闲等状态，这个值将是零；如果还没有执行过任何事件或者从服务器的参数刚被 CHANGE MASTER 或 RESET SLAVE 语句修改过，这个值将是 NULL。
- ❑ Last_IO_Errno
  I/O 线程最近一次发生的错误的出错代码，如果此前没有发生过任何错误，这个值将是零。这个输出列是从 MySQL 5.1.20 版开始引入的。
- ❑ Last_IO_Error
  I/O 线程最近一次发生的出错消息，如果此前没有发生过任何错误，这个值将是空白。从服务器还会把这个输出列的非空白值写入它的出错日志。这个输出列是从 MySQL 5.1.20 版开始引入的。
- ❑ Last_SQL_Errno
  含义和 Last_IO_Errno 输出列相似，但针对的是 SQL 线程。这个输出列是从 MySQL 5.1.20 版开始引入的。
- ❑ Last_SQL_Error
  含义和 Last_IO_Error 输出列相似，但针对的是 SQL 线程。这个输出列是从 MySQL 5.1.20 版开始引入的。

- SHOW STATUS

```
SHOW [GLOBAL | SESSION] STATUS [LIKE 'pattern' | WHERE where_expr]
```

显示服务器的状态变量和它们的值。这些变量提供的信息可以让我们了解服务器的运行状态。12.6.3 节讨论了状态变量的用途和用法。附录 D 对状态变量逐一进行了详细的描述。

如果包含 LIKE 子句，则只显示其名称与给定模式相匹配的变量的信息。如果包含 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

从 MySQL 5.0.2 版开始，MySQL 服务器可以显示全局级（针对整个服务器）或会话级（针对特定客户）的状态变量的值。全局级对应着全体客户，会话级则只对应着当前客户。在默认的情况下，SHOW 语句将显示给定变量在会话级别的值。你可以用一个级别限定符来明确地表明你想查看的是全局级还是会话级的值，如下所示：

```
SHOW GLOBAL VARIABLES;
SHOW SESSION VARIABLES;
```

如果某个变量只有全局级的值，用 GLOBAL 或 SESSION 限定符查看到的值将是一样的。LOCAL 是 SESSION 的一个同义词。

从 MySQL 5.1.12 版开始，你还可以通过查询 INFORMATION_SCHEMA 数据库里的 GLOBAL_STATUS 和 SESSION_STATUS 数据表的办法获得状态变量信息。

- SHOW TABLE STATUS

```
SHOW TABLE STATUS [{FROM | IN} db_name]
```

```
[LIKE 'pattern' | WHERE where_expr]
```

显示关于给定数据表的描述性信息，但只限于你在其上拥有足够权限的数据表。如果包含 LIKE 子句，则只显示其名称与给定模式相匹配的数据表的信息。如果包含一条 WHERE 子句，则只输出那些能够满足给定表达式的数据行。从 MySQL 5.0.1 版开始，这条语句还可以用来查看关于视图的信息，但除了 Name 输出列的值是该视图的名字、Comment 输出列的值是 view 以外，其他输出列的值都将是 NULL。

SHOW TABLE STATUS 语句的输出包含以下输出列。

- Name
  数据表的名字。
- Engine
  存储引擎（MyISAM、InnoDB 等）
- Version
  这个数据表的.frm 文件的版本号。
- Row_format
  数据行的存储格式，如 Fixed（固定长度的数据行）、Dynamic（可变长度的数据行）或 Compressed（压缩存储的数据行）等。
- Rows
  该数据表里有多少行。对于某些存储引擎，比如 InnoDB，这个数字只是一个估算值。
- Avg_row_length
  该数据表各数据行的平均长度，以字节为计量单位。
- Data_length
  该数据表的数据文件的长度，以字节为计量单位。
- Max_data_length
  该数据表的数据文件所能增长到的最大长度。
- Index_length
  该数据表的索引文件的实际长度，以字节为计量单位。
- Data_free
  该数据表的数据文件中尚未使用的字节数。如果这个数字很高，就应该用 OPTIMIZE TABLE 语句来优化这个数据表。
- Auto_increment
  将在该数据表的 AUTO_INCREMENT 数据列里生成的下一个值。
- Create_time
  创建这个数据表的时间。
- Update_time
  最近一次修改这个数据表的时间。
- Check_time
  对于 MyISAM 表，这表示最近一次使用 myisamchk CHECK TABLE 或 REPAIR TABLE 检查或修复这个数据表的时间。如果这个输出列里的值是 NULL，则表明你从没对该数据表进行过检查或修复。
- Collation
  数据表的排序方式。排序方式的名字里隐含着字符集的名字。

- ❑ Checksum
  数据表的校验和。如果此前还没有为它计算过校验和，这个输出列的值将是 NULL。
- ❑ Create_options
  在当初创建数据表的 CREATE TABLE 语句以及随后的 ALTER TABLE 语句里作为 *table_option* 值而给出的额外选项。
- ❑ Comment
  你在创建数据表时给出的任何注释文本。对于 InnoDB 数据表，这个输出列将给出该数据表上的外键定义。在 MySQL 5.1.24 之前，存储这个表的 InnoDB 数据表空间的可用空间量也将显示在这个输出列里。（表可能位于共享表空间中，也可能有自己的表空间。）

● SHOW TABLE TYPES

```
SHOW TABLE TYPES
```

SHOW TABLE TYPES 是 SHOW ENGINES 语句的早期语法。它目前仍可以被识别，但已被淘汰，使用它会导致一条警告信息。请参阅 SHOW ENGINES 语句条目里的相关描述。

● SHOW TABLES

```
SHOW [FULL] TABLES [{FROM | IN} db_name]
  [LIKE 'pattern' | WHERE where_expr]
```

显示给定数据库里的非 TEMPORARY 数据表的名字，但只限于你在其上拥有足够权限的数据表。如果包含一条 LIKE 子句，则只显示其名称与给定模式相匹配的数据表的信息。如果包含一条 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

从 MySQL 5.0.1 版开始，这条语句还将显示视图的名字。从 MySQL 5.0.2 版开始，如果给出了 FULL 关键字，这条语句将为每个输出行标明那是数据表的名字还是视图名字。

这条语句的输出包含以下输出列。

- ❑ Tables_in_*db_name*
  数据表或视图的名字。
- ❑ Table_type
  BASE_TABLE 表示这是一个数据表的名字，VIEW 表示这是一个视图的名字。这个输出列只在你给出了 FULL 关键字的时候才会显示。

● SHOW TRIGGERS

```
SHOW TRIGGERS [FROM db_name] [LIKE 'pattern' | WHERE where_expr]
```

这条语句将显示关于默认数据库里的触发器的信息，如果给出了 FROM 子句，则显示关于给定数据库里的触发器的信息。如果包含 LIKE 子句，则只显示其名称与给定模式匹配的触发器的信息。如果包含 WHERE 子句，则只输出那些能够满足给定表达式的数据行。

SHOW TRIGGERS 语句是从 MySQL 5.0.10 版开始引入的。从 MySQL 5.1.22 版开始，它要求具备 TRIGGER 权限，在那之前要求具备 SUPER 权限。

● SHOW VARIABLES

```
SHOW [GLOBAL | SESSION] VARIABLES [LIKE 'pattern' | WHERE where_expr]
```

显示系统变量和它们的值。这些变量提供了服务器的配置和功能的信息。12.6.1 节讨论了系统变量的用途和用法。附录 D 对系统变量逐一进行了详细的描述。

如果包含 LIKE 子句，则只显示其名称与给定模式相匹配的变量的信息。如果包含 WHERE 子句，

则只输出那些能够满足给定表达式的数据行。

服务器可以显示全局级（针对整个服务器）或会话级（针对特定客户）的系统变量的值。在默认的情况下，SHOW 语句将显示给定变量在会话级别的值；如果会话级别的值不存在，则显示全局级别的值。你可以用一个级别限定符来明确地表明你想查看的是全局级还是会话级的值，如下所示：

```
SHOW GLOBAL VARIABLES;
SHOW SESSION VARIABLES;
```

LOCAL 是 SESSION 的一个同义词。你也可以使用 SELECT 语句去检索某个特定的动态变量的值：

```
SELECT @@GLOBAL.sql_mode, @@SESSION.sql_mode, @@LOCAL.sql_mode;
```

使用 SELECT 语句的好处是可以方便地在特定的上下文里处理查询结果。

从 MySQL 5.1.12 版开始，还可以通过查询 INFORMATION_SCHEMA 数据库里的 GLOBAL_ VARIABLES 和 SESSION_VARIABLES 数据表来获得系统变量信息。

- SHOW WARNINGS

```
SHOW WARNINGS [LIMIT [skip_count,] show_count]

SHOW COUNT(*) WARNINGS
```

SHOW WARNINGS 语句将显示因为最近一条语句的执行而导致的出错消息、警告消息或其他提示性消息。如果最近一条语句执行成功，SHOW WARNINGS 语句将返回一个空集。

SHOW COUNT（*）WARNINGS 语句显示保存在 warning_count 系统变量里的消息计数值。（还有一个与此相关的 error_count 变量，但只统计出错消息的个数。）warning_count 变量的值有可能大于 SHOW WARNING 语句所显示的消息的个数，这是因为被保存起来供 SHOW WARNING 语句显示的消息个数受限于系统变量 max_error_count 的设置值，而 warning_count 变量里的计数值是曾经发生过的上述几种消息——不管它们是否还被保存着——的总个数。

LIMIT 子句可以用来限制 SHOW WARNING 语句返回的输出行的个数，该子句的语句和 SELECT 语句的 LIMIT 子句完全一样。

- START SLAVE

```
START SLAVE [slave_option [, slave_option] ...]

START SLAVE [SQL_THREAD] UNTIL
  MASTER_LOG_FILE = 'file_name', MASTER_LOG_POS = position

START SLAVE [SQL_THREAD] UNTIL
  RELAY_LOG_FILE = 'file_name', RELAY_LOG_POS = position
```

这条语句和 STOP SALVE 语句控制着从服务器上复制线程的操作。不带任何选项的 START SLAVE 语句将同时启动从服务器的 I/O 线程和 SQL 线程，不带任何选项的 STOP SLAVE 语句将同时中止这两个线程。可选的 *slave_option* 值用来表明启动或中止哪一个线程。

- ❑ IO_THREAD

  启动或中止 I/O 线程，该线程负责从主服务器读取事件并把它们写入中继日志。

- ❑ SQL_THREAD

  启动或中止 SQL 线程，该线程负责从中继日志读出事件并执行。

如果没有指定任何线程或只给出了 SQL_THREAD 选项，UNTIL 子句就可以派上用场了。根据这个子句所给出的日志文件和位置选项，从服务器在启动后将一直运行到它的 SQL 线程到达主服务器的二

进制日志或是从服务器的中继日志里的指定位置为止。如果 SQL 线程已经在运行，从服务器将忽略 UNTIL 子句并生成一条警告消息。如果该子句包含 SQL_THREAD 选项，从服务器将只启动 SQL 线程；否则，同时启动 I/O 和 SQL 两个线程。

- START TRANSACTION

```
START TRANSACTION [WITH CONSISTENT SNAPSHOT]
```

开始一个事务：禁用 autocommit 模式，直到遇见下一条 COMMIT 或 ROLLBACK 语句为止。在 autocommit 模式被禁用期间执行的语句将作为一个整体被提交或回滚。

在事务被提交或回滚之后，autocommit 模式将恢复到它在 START TRANSACTION 语句开始执行之前的状态。你可以使用 SET autocommit 语句明确地对 autocommit 模式作出设定。对 autocommit 变量的详细描述见附录 D。

如果带有 WITH CONSISTENT SNAPSHOT 子句，这条语句将为各有关数据库拍一个定格快照供事务里的语句使用。对 InnoDB 存储引擎而言，因为这个子句并不改变当前的隔离级别，所以它只在隔离级别是 REPEATABLE READ 和 SERIALIZABLE 的情况下才真正有效。对 Falcon 存储引擎而言，不管当前的隔离级别是什么，这个子句都会使用 REPEATABLE READ 隔离级别来提供一张定格快照。

START TRANSACTION 语句会隐含地把当前客户此前通过 LOCK TABLE 语句获得、但尚未释放的数据表锁全都释放掉。在某个事务的进行过程中执行一条 START TRANSACTION 语句将导致该事务被隐含地提交。

- STOP SLAVE

```
STOP SLAVE [slave_option [, slave_option] ...]
```

这条语句和 START SALVE 语句控制着从服务器上复制线程的操作，详见 START SLAVE 语句条目里的描述。

- TRUNCATE

```
TRUNCATE [TABLE] tbl_name
```

这条语句将丢弃并重新创建一个数据表以达到快速清空其内容的目的，这显然要比一行一行地删除快得多。从 MySQL 5.1.16 版开始，必须具备 DROP 权限才能使用这条语句，此前的版本要求你必须具备 DELETE 权限。

对于 InnoDB 存储引擎，这条语句在 MySQL 5.0.3 版之前被实现为 DELETE FROM *tbl_name*。从 MySQL 5.0.3 版开始，InnoDB 存储引擎直接实现了这种快速清空的操作。

这条语句不支持事务处理，在事务过程中或者在持有任何显式数据表锁期间发出一条 TRUNCATE TABLE 语句将导致一个错误。

- UNION

```
select_stmt
  UNION [DISTINCT | ALL] select_stmt
  [UNION [DISTINCT | ALL] select_stmt] ...
  [ORDER BY col_list] [LIMIT [skip_count,] show_count]
```

UNION 用来把多条 SELECT 语句的检索结果合并在一起，而每条 SELECT 语句必须在它自己的结果集里生成出同样个数的输出列。在最终结果里，各输出列的名字将根据第一条 SELECT 语句里的数据列的名字来确定，各输出列的数据类型将综合考虑从所选数据表相应数据列的全体数据值的情况来确定。

可以在 UNION 关键字之后加上一个 DISTINCT 关键字来剔除重复的输出行，也可以加上一个 ALL 关键字保留它们以返回所有被选取出来的数据行。如果 DISTINCT 和 ALL 关键字都没有给出，默认的行为是剔除重复的输出行。DISTINCT 方式的 UNION 操作（无论隐式或显式）将使它左侧的所有 ALL 关键字无效化：

```
mysql> SELECT 1 UNION ALL SELECT 2 UNION ALL SELECT 1;
+---+
| 1 |
+---+
| 1 |
| 2 |
| 1 |
+---+
mysql> SELECT 1 UNION ALL SELECT 2 UNION SELECT 1;
+---+
| 1 |
+---+
| 1 |
| 2 |
+---+
```

如果要让 SELECT 语句带有 ORDER BY 和 LIMIT 子句，就必须把每一个 SELECT 语句都用圆括号括起来。（如果某个 SELECT 语句带有 ORDER BY 子句，该子句将只在同时带有 LIMIT 子句的时候才会执行，它让 LIMIT 子句对排序后的数据行进行筛选。ORDER BY 子句对最终 UNION 结果里的输出行的先后顺序没有影响。）如果需要把 UNION 语句的最终结果当做一个整体而对之使用 ORDER BY 或 LIMIT 子句，必须先把每一条 SELECT 语句用圆括号括起来，再把那条 ORDER BY 或 LIMIT 子句写在最后一个右括号的后面。ORDER BY 子句中的数据列只允许引用第一个 SELECT 语句里的数据列的名字。

- UNLOCK TABLE

```
UNLOCK {TABLE | TABLES}
```

这条语句将释放当前客户所持有的所有数据表锁。

如果客户在断开服务器连接时还持有未释放的数据表锁，MySQL 服务器将在关闭该连接时释放它们。

如果当前客户在持有数据表锁的情况下开始了一个事务，MySQL 服务器将隐式地释放那些数据表锁。

- UPDATE

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
  SET col_name=expr [, col_name=expr] ...
  [WHERE where_expr] [ORDER BY ...] [LIMIT n]

UPDATE [LOW_PRIORITY] [IGNORE] tbl_refs
  SET col_name=expr [, col_name=expr] ...
  [WHERE where_expr] [ORDER BY ...] [LIMIT n]
```

UPDATE 语句的第一种语法将对数据表 *tbl_name* 里的现有数据行的内容进行修改。UPDATA 语句的第二种语法类似于第一种，但允许一次给出多个数据表以进行一次多数据表修改。*tbl_refs* 项的语法和 SELECT 语句里的相似，二者的区别是这里不再允许使用子查询来充当数据表。

只有用 WHERE 子句所给出的表达式筛选出来的那些数据行才会被修改。对于选出来的每一个数据行，在 SET 子句里给出的每一列将被设置为相应的表达式的值。

```
UPDATE member SET expiration = NULL, phone = '197-602-4832'
  WHERE member_id = 14;
```

WHERE 子句允许包含子查询，但不允许子查询从正被修改的数据表选取数据行。

如果没有给出任何 WHERE 子句，数据表里的所有数据行都将被修改！

在默认的情况下，UPDATE 语句的返回值是它实际修改的数据行的个数。如果某个数据行的数据列在修改前后没有发生任何实质性变化，UPDATA 语句将不把它统计在内。如此说来，把某个数据列设置为它的当前值不算影响了数据行。如果真的需要 UPDATE 语句返回一个数值告诉你到底有多少个数据行和它的 WHERE 子句相匹配（不管它实际修改了多少个数据行），你应该在与服务器建立连接时给出 CLIENT_FOUND_ROWS 选项，详见附录 G 里关于 mysql_real_connect()函数的条目。

LOW_PRIORITY 选项将导致 UPDATE 语句延迟到没有任何客户读取该数据表的时候才实际执行。这个选项只适用于支持数据表级锁定机制的存储引擎，如 MyISAM、MEMORY 和 MERGE。

如果修改某个数据行会导致某个唯一化索引出现重复的键值，UPDATE 语句将报告出错并中止执行，不再修改剩余的数据行。给 UPDATE 语句加上 IGNORE 关键字将使它跳过这样的数据行继续执行，而且不会报告出错。在严格模式下，IGNORE 关键字还将使得会导致中止执行的数据转换错误被当做一个不那么致命的警告来处理——把导致出错的数据列修改为最接近的合法值。

如果带有 ORDER BY 子句，UPDATE 语句将按给定顺序依次修改数据行。这个子句和 SELECT 语句的同名子句有同样的语法。

如果给出了 LIMIT 子句，*n* 值设定的是将被修改的数据行的最大个数。

对于一条多数据表 UPDATE 语句，WHERE 子句允许基于数据表之间的关联指定条件，它的 SET 子句允许对多个数据表里的数据列进行修改。比如说，下面的语句将修改 t1 数据表里的 id 值与 t2 数据表里的 id 值相匹配的那些数据行，把 quantity 值从 t2 数据表复制到 t1 数据表：

```
UPDATE t INNER JOIN t2 SET t.quantity = t2.quantity WHERE t.id = t2.id;
```

- USE

```
USE db_name
```

把名为 *db_name* 的数据库设为默认数据库，如果你在引用某个数据表、视图或存储程序时没有明确地给出数据库，它们将默认来自该数据库。如果 USE 语句执行成功，MySQL 服务器将把会话级 character_set_database 和 collation_database 系统变量设置为数据库级字符集和排序方式。

如果给定数据库不存在或者是你没有足够的权限去访问它，USE 语句将执行失败。

E.2　复合语句的语法

本节描述用来编写复合语句的各种语句的语法。复合语句由关键字 BEGIN 和 END 之间的一条或多条语句构成，它们的主要用途是编写将被保存在服务器端的存储程序（函数、过程、触发器和事件）。

程序体内部的每条语句都必须以分号（;）字符结尾。如果打算使用 mysql 程序来创建一个包含有多条语句的存储例程，必须临时修改 mysql 程序的语句分隔符，确保 mysql 程序本身不会去解释那些“;”字符。可以用 delimiter 命令来做这件事情，但一定要保证新挑选的语句分隔符不会出现在用来定义存储例程的语句里。如下所示：

```
mysql> delimiter $
mysql> CREATE FUNCTION myfunc ()
    -> RETURNS INT DETERMINISTIC
    -> BEGIN
```

```
    -> DECLARE i INT;
    -> DECLARE j INT;
    -> SET i = 2;
    -> SET j = 4;
    -> RETURN i * j;
    -> END$
mysql> delimiter ;
mysql> SELECT myfunc();
+----------+
| myfunc() |
+----------+
|        8 |
+----------+
```

关于定义存储程序的更多信息，请参阅 4.1 节。

E.2.1　流程控制语句

本节里的语句用来把一组语句归聚为一个语句块并提供各种流程控制构造。在这些语句的语法表达式里，每个 *stmt_list* 代表一个由一条或多条语句构成的语句块，每条语句都以一个分号字符（;）结束。

有些构造可以带有标签（BEGIN、LOOP、REPEAT 和 WHILE）。标签不区分字母的大小写，但必须遵守以下规则。

- ❑ 如果在某个构造的开头给出了一个标签，在这个构造的末尾还可以再给出一个与之同名的标签。
- ❑ 如果在某个构造的开头没有给出标签，在这个构造的末尾就不允许给出与之同名的标签。

- BEGIN ... END

```
BEGIN [stmt_list] END
label: BEGIN [stmt_list] END [label]
```

BEGIN...END 构造将创建一个语句块，可以包含多条语句。如果某个存储例程的程序体需要包含多条语句，就必须把它们放在一个 BEGIN 块里。此外，如果存储例程还包含有 DECLARE 语句，它们将只能出现在 BEGIN 块的开头部分。

- CASE

```
CASE [expr]
  WHEN expr1 THEN stmt_list1
  [WHEN expr2 THEN stmt_list2] ...
  [ELSE stmt_list]
END IF
```

CASE 语句提供了一种分支型的流程控制构造。如果给出了初始表达式 *expr*，CASE 语句将把它与每个 WHEN 关键字后面的表达式比较。在找到第一个匹配时，相应的 THEN 关键字后面的 stmt_list 将被执行。这适合用来比较给定的值与一组值。

如果没有给出初始表达式 *expr*，CASE 语句将依次对每一个 WHEN 表达式求值。在遇到第一个取值为 true 的 WHEN 表达式时，相应的 THEN 关键字后面的 *stmt_list* 将被执行。这适合用来进行“不等于”比较或是测试任意条件。

如果没有找到匹配的 WHEN 表达式，ELSE 关键字后面的 *stmt_list*（如果有的话）将被执行。

请注意，这里介绍的 CASE 语句不同于 C.1.4 节里介绍的 CASE 操作符。

- IF

```
IF expr1 THEN stmt_list1
  [ELSEIF expr2 THEN stmt_list2] ...
```

```
  [ELSE stmt_list]
END IF
```

IF 语句提供了一种分支型的流程控制构造。如果 IF 关键字后面的表达式为真，第一个 THEN 关键字后面的 *stmt_list* 将被执行。否则，依次对各 ELSEIF 关键字后面的表达式进行求值。在遇到第一个取值为 true 的 ELSEIF 表达式时，相应的 THEN 关键字后面的 *stmt_list* 将被执行。如果所有表达式的求值结果都不是 true，ELSE 关键字后面的 *stmt_list*（如果有的话）将被执行。

请注意，这里介绍的 IF 语句与在 C.2.1 节里介绍的 IF()函数是不同的概念。

- ITERATE

```
ITERATE label
```

ITERATE 语句用来开始下一次循环，它只能在循环构造内部使用。它可以出现在 LOOP、REPEATE 和 WHILE 语句内。

- LEAVE

```
LEAVE label
```

LEAVE 语句用来退出一个带有给定标签的流程控制构造。该语句只能出现在带有给定标签的构造的内部。

- LOOP

```
LOOP stmt_list END LOOP
label: LOOP stmt_list END LOOP [label]
```

这条语句用来创建一个执行循环。循环内的语句将反复执行，直到控制权被转出该循环。

- REPEAT

```
REPEAT stmt_list UNTIL expr END REPEAT
label: REPEAT stmt_list UNTIL expr END REPEAT [label]
```

这条语句用来创建一个执行循环。循环内的语句将反复执行，直到表达式 *expr* 的值为 true 为止。

- RETURN

```
RETURN expr
```

RETURN 语句只能在存储函数里使用，不能在存储过程、触发器或事件里使用。一旦执行 RETURN 语句，将结束存储函数的执行，而表达式 *expr* 的值将被传递给发出这次函数调用的语句。在同一个存储函数里可以有多条 RETURN 语句，至少有一条。

- WHILE

```
WHILE expr DO stmt_list END WHILE
label: WHILE expr DO stmt_list END WHILE [label]
```

这条语句用来创建一个执行循环。循环内的语句将反复执行，只要表达式 *expr* 的值为 false。

E.2.2　声明语句

DECLARE 语句用来声明局部变量、条件、游标（cursor）和处理程序（handler）。

- DECALRE

```
DECLARE var_name [, var_name] ... type [DEFAULT value]

DECLARE condition_name CONDITION FOR named_condition
```

```
named_condition: {SQLSTATE [VALUE] sqlstate_value | mysql_errno}

DECLARE cursor_name CURSOR FOR select_stmt

DECLARE handler_type
  HANDLER FOR handler_condition [, handler_condition] ...
  statement
handler_type: {CONTINUE | EXIT}

handler_condition:
    SQLSTATE [VALUE] sqlstate_value
  | mysql_errno
  | condition_name
  | SQLWARNING
  | NOT FOUND
  | SQLEXCEPTION
```

以上是声明局部变量、条件、游标和处理程序的语法。DECALRE 语句只能出现在 BEGIN 块的开头。如果有多条 DECALRE 语句，它们必须按以下顺序出现。

- 变量和条件声明
- 游标声明
- 处理程序声明

如果在 DECALRE 关键字的后面以逗号为分隔符列出一组变量，就能把那些变量声明为可以在当前例程里使用的局部变量。局部变量只能在声明它们的 BEGIN 块里以及嵌套在该 BEGIN 块内部的其他语句块里使用。

在 DECALRE 语句里还可以用 DEFAULT 子句对局部变量进行初始化。如果没有 DEFAULT 子句，初始值将是 NULL。如果需要在例程里把值赋给局部变量，可以使用一条 SET 语句或是一条 SELECT...INTO *var_name* 语句。

DECLARE...CONDITION 语法用来为条件创建名字。如此创建的名字可以在 DECLARE...HANDLER 语句里引用。*named_condition* 可以是一个由括在引号里的 5 个字符构成的 SQLSTATE 值，也可以是一个数值形式的 MySQL 专用出错代码。

DECLARE...CURSOR 语法用来声明与给定 SELECT 语句相关联的游标，那条 SELECT 语句不得包含 INTO 子句。游标可以用 OPEN 语句打开，可以在 FETCH 语句里用来检索数据行、可以用 CLOSE 语句关闭。

DECLARE...HANDLER 语法用来把一个或多个条件与一条语句关联在一起，只要那些条件里有一个为真，该语句就将被执行。*handler_type* 值决定着被关联语句执行完毕后将发生什么事情。如果 *handler_type* 值是 CONTINUE，继续执行下一条语句；如果是 EXIT，立刻退出当前 BEGIN 块。

handler_condition 可以是任何一种以下类型的值。

- 由括在引号里的 5 个字符构成的 SQLSTATE 值。这个值不应该是'00000'，因为那代表着“成功”而不是代表着“发生了一个错误”。
- 数值形式的 MySQL 专用出错代码。这个值不应该是零，因为那代表着“成功”而不是代表着“发生了一个错误”。
- 此前用 DECLARE...CONDITION 语句声明了一个名字的条件。
- SQLWARNING，这个值将匹配所有以 01 开头的 SQLSTATE 值。
- NOT FOUND，这个值将匹配所有以 02 开头的 SQLSTATE 值。
- SQLEXCEPTION，这个值将匹配所有与 SQLWARNING 或 NOT FOUND 不匹配的 SQLSTATE 值。

E.2.3　游标语句

本节里的语句可以用来打开或关闭游标、使用已经打开的游标来检索数据行。就目前而言，游标都是只读的，并且是每次只能移动到结果集里的下一行（换句话说，无法利用游标移动到结果集里的上一个数据行）。

- CLOSE

```
CLOSE cursor_name
```

关闭给定的游标，它在被关闭前必须是打开的。当一个 BEGIN 块结束时，在这个 BEGIN 块里声明的游标都将自动关闭。

- FETCH

```
FETCH [[NEXT] FROM] cursor_name INTO var_name [, var_name] ...
```

利用给定游标把结果集里的下一行提取到在 INTO 关键字后面列出的变量里，该游标必须是已经打开的。如果已经没有数据行可供提取，将导致一个 SQLSTATE 值是 02000 的错误。

- OPEN

```
OPEN cursor_name
```

打开给定的游标以便在后面的 FETCH 语句里使用它。

E.3　注释语法

MySQL 允许在 SQL 代码里穿插一些注释。注释的基本用途是对保存在文件里的语句进行说明。本节描述如何在 SQL 语句里写出注释。

MySQL 服务器支持的注释类型有 3 种。

- 从字符“#”开始一直到行尾的所有文字都将被视为注释内容。这种语法与绝大多数 Unix shell 以及许多种脚本编程语言如 Perl、PHP 或 Ruby 等使用的注释语法相同。

```
# this is a single line comment
```

- “/*”和“*/”之间的所有文字都将被视为注释内容。这种形式的注释允许跨越多行。这种语法与 C 语言所使用的注释语法相同。

```
/* this is a single line comment */
/* this
   is a multiple line
   comment
*/
```

- 可以用两个连字符加一个空格（“-- ”）或者两个连字符加一个控制字符（如换行符）来开始一条注释。从双连字符到行尾的所有文字都将被视为注释内容。

```
--
-- This is a comment
--
```

MySQL 的双连字符注释风格与标准 SQL 的注释风格不太一样，后者只要求以两个连字符开头，不要求后面必须有一个空格。MySQL 要求在连字符的后面必须加上一个空格是为了避免二义性，否则“5--7”（5 减去－7）这样的表达式就很可能被误认为包含一条注释。因为人们不太可能写出“5-- 7”这样的表达式，所以 MySQL 的要求还是有实用意义的，但也仅此而已。如果你打算从其他数据库系

统把包含双连字符注释 SQL 代码移植到 MySQL 里来，一定要保证注释不违反 MySQL 的有关规定。

在执行语句时，服务器会忽略其中的注释，但以“/*!”开始的 C 语言风格注释却是个例外。我们可以把 MySQL 独有的关键字“隐藏”在以“/*!”开头（请注意，不是以“/*”开头）的 C 风格注释里。在遇到这种特殊形式的注释时，MySQL 将识别出其中的关键字并执行相应的动作，而其他的数据库服务器却会把它们当做注释的一部分忽略掉。这种安排增加了代码的可移植性，至少对支持 C 语言风格注释的其他服务器来说是这样的：当代码在 MySQL 环境里执行时，MySQL 独有的函数将发挥作用；当需要把代码拿到其他数据库环境里去使用时，它们也用不着修改。下面两条语句在其他数据库服务器看来是等价的，但 MySQL 在遇到第二条语句时却会执行 INSERT DELAYED 操作：

```
INSERT INTO mytbl (id,date) VALUES(13,'2008-09-28');
INSERT /*! DELAYED */ INTO mytbl (id,date) VALUES(13,'2008-09-28');
```

C 语言风格的注释还可以用来给语句加上版本控制信息，即通过在“/*!”序列的后面加上 5 个数字的版本号来达到这样一个效果：如果 MySQL 服务器低于该注释所给出的版本号，它将忽略这条注释。请看下面这条 SHOW STATUS 语句，如果 MySQL 服务器不是 5.0.2 或者更高的版本，它将忽略其中的注释（只有 5.0.2 或更高版本的 MySQL 服务器才理解 GLOBAL 和 SESSION 关键字）：

```
SHOW /*!50002 GLOBAL */ STATUS;
```

附录 F

MySQL 程序指南

我们将在本附录里介绍调用 MySQL 程序的基本信息，并详细讨论下列程序。我们将在后面依次介绍每一个程序的用途、执行语法、它所支持的选项以及其内部使用的各种变量。如无特别注明，本附录所列举的选项和变量至少从 MySQL 5.0.0 版本开始就已经存在于 MySQL 软件中了。

- myisamchk
  用来检查和修复 MyISAM 数据表、分析键分布情况、禁用或启用索引的工具程序。
- myisampack
  用来对数据表进行压缩并生成只读 MyISAM 数据表的工具程序。
- mysql
  具备命令行编辑功能、用来向 MySQL 服务器发送 SQL 语句的交互式程序。你也可以用它来以批处理方式执行存放在文件里的语句。
- mysql.server
  用来启动和中止 MySQL 服务器的脚本。
- mysql_config
  当你准备编译一个基于 MySQL 的程序时，可以利用这个工具程序来确定该程序的标志。
- mysql_install_db
  用来对服务器的数据目录和权限表进行初始化的脚本。
- mysqladmin
  用来完成管理工作的工具程序。
- mysqlbinlog
  以文本格式显示二进制文件和中断日志内容的程序。
- mysqlcheck
  一个用来对数据表进行检查、修复、优化和分析的工具程序。
- mysqld
  MySQL 服务器。只有先运行了这个程序，客户程序才能访问在它管理之下的数据库。
- mysqld_multi
  用来同时启动和关闭多个服务器的脚本。
- mysqld_safe
  用来启动和监控 MySQL 服务器的脚本。
- mysqldump
  用来导出数据表内容的客户程序。

- mysqlhotcopy
  数据库备份实用工具程序。
- mysqlimport
  用来往数据表里批量加载数据的客户程序。
- mysqlshow
  用来查看关于数据库或数据表的信息的客户程序。
- perror
  显示出错代码含义的实用工具程序

在后面的内容里，我们将把语法说明中的可选信息放在方括号（[]）里。

F.1 查看程序的帮助信息

在介绍每一个程序的时候，我们将把它目前所支持的所有选项全都列举出来。如果某个程序识别不出本附录列举的某个选项，那很可能是因为你使用的程序是一个比较老的版本，那个“肇事”选项还没有添加进来。

如果想知道某个程序支持哪些选项，可以去查看该程序的帮助信息，这是从程序本身获取信息的快速办法。具体地说，对于 MySQL 服务器程序（mysqld），用--version 和--help 选项来启动它；对于其他程序，用--help 选项来运行它。比如说，如果拿不准应该如何使用 mysqlimport 程序，可以用下面这条命令来调用它：

```
% mysqlimport --help
```

-?选项和--help 选项的作用是一样的，但你的 shell（命令解释器）有可能会把“?”字符解释为一个文件名通配符：

```
% mysqlimport -?
mysqlimport: No match.
```

如果遇到这种情况，请试试下面这条命令：

```
% mysqlimport -\?
```

有些选项只在特定条件下才会被列举在帮助信息里。比如说，与 SSL 有关的选项只有在你把 SSL 支持的功能事先编译到 MySQL 软件里的情况下才会出现在帮助信息里；而 Windows 系统所独有的选项（比如--pipe）也只有在 Windows 系统上才会出现在帮助信息里。

MySQL 程序的帮助信息还将显示它在默认的情况下会到什么地方读取选项文件、它支持哪些变量，等等。

F.2 设置程序选项

大多数 MySQL 程序都有一些能够影响其操作行为的选项。这些选项既可以在命令行上给出，也可以在选项文件里给出。此外，有一些选项可以通过设置环境变量来设定。在命令行上给出的选项优先于以其他方式给出的选项，在选项文件里给出的选项又优先于通过环境变量设置的选项。

大多数选项都有一个长（完整单词）形式和一个短（单字符）形式。刚才见到的--help 和-?就是一个典型的例子。后面跟有设置值的长格式选项要以--*name* = *val* 或--name val 的格式给出，其中的 *name* 是选项的名字，*val* 是该选项的值。在大多数场合，如果短格式选项的后面还有一个值，

选项和值之间允许出现空白。比如说，当你给出一个用户名时，-usampadm 和-u sampadm 是等价的。-p（口令）选项是个例外，它是可选的，但如果给出了，它和它后面的口令值之间不允许有任何间隔。

选项名称区分字母大小写。例如，myisamchk 程序支持--help 和--HELP，但这二者稍有不同。

选项值字母可以区分大小写，也可以不区分大小写。例如，用户名和口令是区分大小写的，但--protocol 选项的值则不区分。为了进行 TCP/IP 连接，--Protocol=tcp 与--Protocol=TCP 是等价的。

许多选项为布尔型，设置值为开/关形式。这种选项有一个基本形式和一组标准的相关形式，如表 F-1 所示。

表 F-1

选　　项	含　　义
--*name*	基本形式，启用该选项
--enable-*name*	--enable-前缀，启用该选项
--disable-*name*	--disable-前缀，禁用该选项
--skip-*name*	--skip-前缀，禁用该选项
--*name* = 1	= 1后缀，启用该选项
--*name* = 0	= 0后缀，启用该选项

比如说，许多客户程序都支持启用客户/服务器协议中的压缩功能这个选项。如果你给出了--compress 选项，则启用压缩功能；如果没有给出，则禁用压缩功能。也可以采用其他方式，--enable-compress 和--compress=1 也可以启用压缩功能，而--disable-compress、--skip-compress 和--compress =0 将被解释为不使用压缩功能。

对于默认禁用的选项，显式禁用选项的格式非常有用。对于协议压缩，只要不给出--compress 选项就能禁用它。但这对于默认启用的选项就不适用。例如，mysqldump 的--quote-name 选项是默认启用的，你省略它并不能禁用名字引用，但可以通过指定--skip-quote-names、--disable-quote-names 或--quote-names=0 来完成。

本附录描述程序时使用了"布尔"来表明哪个选项优先解释，也就是说，在表中显示了前缀和后缀的选项是受支持的。

如果有疑问，可以查看程序的帮助消息，了解受支持的是哪个选项（参见 F.1 节）。

MySQL 程序还有其他一些标准的选项处理属性。

- 长选项可以截短为无歧义的前缀，这样可以更容易地指定名字很长的选项。如果指定的前缀长度不足以消除歧义，你调用的程序将会告诉你，并且列出与前缀匹配的选项。

```
% mysql --h
mysql: ambiguous option '--h' (help, html)
```

- 可以从命令行或者在选项文件里设置程序变量，变量名将被当做选项名来对待。这方面的详细讨论请参见 F.2.1 节的第 2 小节。
- --loose-前缀能使来自不同版本的程序对选项的格式不那么挑剔。比如说，4.1 及更高版本的 MySQL 服务器都能识别--old-passwords 选项，但老版本就不行了。如果把这个选项以--loose-old-passwords 的形式给出，那么，4.0.2 及更高版本的服务器就能根据它自己是否支持--old-passwords 选项而使用或者忽略这个选项。应用--loose 后，不能识别的选项只会引发一个警告，不会导致程序中止，并报告出错。

- mysqld 程序还支持使用--maximum-前缀来给用户定义变量设置最大值。比如说，服务器允许用户通过改变 sort_buffer_size 变量值来调整排序缓冲的尺寸。如果你想把这个变量的最大值设置为 64 MB，就需要在启动服务器时给出--maximum-sort_buffer_size=64MB 选项。

F.2.1　MySQL程序的标准选项

有些选项在各种 MySQL 程序里都有着同样的含义和作用，我们把这些选项称为 MySQL 程序的“标准”选项。为简洁起见，我们决定在这里对它们做一次全面的介绍，等介绍到某个具体的程序时，我们将只在某些小节里列出该程序所支持的标准选项，而不再重复介绍它们的用途和用法了。本小节只列出了长格式名称，如无特别说明，你完全可以使用相应的短格式选项来运行程序。

下面是这些标准选项，如果 MySQL 在编译时没有重新配置，应用的就是默认选项。

- --character-sets-dir = *dir_name*
  用来存放字符集文件的目录。
- --compress 或 -C （布尔）
  使用客户/服务器通信协议中的压缩功能——如果服务器支持的话。这个选项只能由客户程序使用。
- --debug = *debug_options* 或 -# *debug_options*
  打开调试输出。如果 MySQL 在编译时没有启用调试支持机制，这个选项将不可用。*debug_options* 是由一个或者多个以冒号分隔的选项构成的字符串。它的常见设置值是 d:t:o, *file-name* 表示启用调试功能，打开进入和退出函数调用时的跟踪机制并把输出发送到文件 *file_name*。
  如果要进行很多调试工作，应该查阅 DBUG 库用户手册，了解可用的所有选项。这个手册位于 MySQL 源发行版本的 dbug 文件。
- --debug-check（布尔）
  程序执行时检查内存和已打开文件的使用情况。
- --debug-info（布尔）
  类似于--debug-check，但还显示有关内存和 CPU 利用的信息。
- --default-character-set = *charset*
  默认字符集的名字。
- --help 或 -?
  显示帮助信息并退出。参见 F.1 节。
- --host=*host_name* 或 -h *host_name*
  将要连接的主机（即 MySQL 服务器在其上运行的主机）。这个选项只能由客户程序使用，默认值为 localhost。
- --password [= *pass_val*] 或 -p[*pass_val*]
  用来连接 MySQL 服务器的口令。如果没有在这个选项名的后面给出 *pass_val*，程序将提示你输入。如果给出了 *pass_val*，它必须紧跟在选项名的后面，中间不留任何空隙。也就是说，这个选项的短格式必须写成-p *pass_val*，不能写成-p *pass_val*。这个选项只能由客户程序使用。
- --pipe 或 -W
  使用一个命名管道来连接服务器。这个选项只能由在 Windows 系统中运行的客户程序使用，

而且只能用来连接那些支持使用命名管道的基于 Windows 的服务器。

- `--port = port_num` 或 `-P port_num`

  对于 mysqld，这是它监听 TCP/IP 连接时使用的端口号。默认端口号是 3306。对于客户程序，这个选项指定的是通过 TCP/IP 连接服务器的端口。

- `--protocol=protocol-type`

  这个选项只能用于客户程序，它指定与服务器建立的连接类型。*protocol-type* 可以为 `tcp`（使用 TCP/IP）、`socket`（使用 Unix 套接字文件）、`pipe`（使用 Windows 命名管道）或 `memory`（使用共享内存）。这个值不区分大小写。

  有些连接类型是特定于平台的，或者只对本地服务器连接有用（服务器运行的主机与客户程序相同）。

  - 套接字、命名管道和共享内存连接只用于与本地服务器的连接。
  - 套接字连接只用于 Unix。
  - 命名管道和共享内存只用于 Windows。
  - TCP/IP 连接可用于任何平台，用于本地或远程服务器的连接。

  `--protocol` 选项可与指定连接方式的其他选项结合使用。

  - 对于 TCP/IP 连接，可以使用`--host` 和`--port` 选项指定主机名和 TCP/IP 端口号。
  - 对于套接字和命名管理连接，可以使用`--socket` 选项指定 Unix 上的 Unix 套接字文件名或 Windows 上的命名管道名称
  - 对于共享内存连接，可以使用`--share-memory-base-name` 选项指定共享内存名称。

- `--set-variable var=value` 或 `-O var=value`

  对程序操作参数进行赋值。`var` 是变量名，`value` 是该变量的值。这个选项已被弃用，参见 F.2.1 节的第 2 小节。

- `--shared-memory-base-name=name`

  用于进行共享内存连接的共享内存的名字。默认值为 `MYSQL`。该值是区分大小写的。

- `--silent` 或 `-s`

  在沉默模式里运行。这并不意味着程序完全沉默，但程序在这种模式下产生的输出信息的确要比在正常情况下少。有些程序允许多次给出这个选项，这将使程序更加沉默。

- `--socket = file_name` 或 `-S file_name`

  对于 Unix 系统上的客户程序，这是它在连接主机 `localhost` 上的服务器时使用的 Unix 套接字文件的路径名。默认的 Unix 套接字文件名是/tmp/mysql.sock 如果 MySQL 主机上的文件名区分大小写，那么路径名也区分大小写。对于 Windows 系统上的客户程序，这是它通过命名管道连接 MySQL 服务器时使用的命名管道的名字。默认的管道名称是 `MySQL`，不区分大小写。

- `--user = user_name` 或 `-u user_name`

  对于 mysqld 客户程序，这是它在连接服务器时使用的 Unix 账户的名字或用户 ID。不过，要想让这个选项有效果，MySQL 服务器必须是以根用户 `root` 身份启动的才行，因为只有这样它才能改变自己的用户 ID。对于客户程序，这个选项指定了用来运行服务器的 MySQL 用户名。默认值是登录名（Unix 下）或 ODBC（Windows 下）。

- `--verbose` 或 `-v`

  在详细信息模式里运行程序，程序将比正常情况产生更多的输出。有些程序允许你重复多次地给出这个选项，这将使程序产生更多的输出信息。

- --version 或 -V
  让程序显示其版本信息字符串并退出。

1. SSL标准选项

下列选项用来建立安全连接，只有在 MySQL 软件里编译有 SSL 支持机制时才能使用。与建立安全连接有关的详细讨论见 13.3 节。

- --ssl（布尔）
  允许 SSL 连接。与 SSL 有关的其他选项都隐含着--ssl 选项功能。更常用的是--skip-ssl，表示禁用 SSL 连接。
- --ssl-ca = *file_name*
  颁证机构的证书文件的路径名。
- --ssl-capath = *dir_name*
  用来保存信任证书的子目录的路径名，用于证书验证。
- --ssl-cert = *file_name*
  证书文件的路径名。
- --ssl-cipher = *str*
  这个字符串给出的是用来对客户/服务器之间的通信进行加密的 SSL 加密类型的名称。这个字符串应该给出一个或者多个以逗号分隔的加密类型的名称。
- --ssl-key = *file_name*
  密钥文件的路径名。
- --ssl-verify-server-cert（布尔）
  这个选项只应用于客户程序，命令客户检查从服务器收到的证书的 Common Name 值。如果这个值不同于客户连接的主机，连接就被放弃，这个值是从 MySQL5.0.23/5.1.11 开始引入的。

2. 设置程序变量

部分 MySQL 程序有一些允许用户设置的变量（操作参数）。你可以像对待程序选项（即把变量名视为选项名）那样设置变量。比如说，如果你想在启动 mysql 程序时把 connect_timeout 变量设置为 10，可以使用如下所示的命令：

```
% mysql --connect_timeout=10
```

这种语法还允许你把变量名中的下划线字符（_）写成连字符（-），这就使变量选项与程序选项更相似了：

```
% mysql --connect-timeout=10
```

对于表示缓冲尺寸或长度的变量，如果指定为不带后缀的数字，则其值以字节为单位；也可以指定后缀为 K、M、G，分别表示单位是千字节、兆字节和吉字节。后缀不区分大小写，所以也可以写成 k、m、g。

使用--set-variable 选项(或者它的短格式-O)来设置变量。用这种语法设置 Connect.timeout 的命令如下所示：

```
% mysql --set-variable=connect_timeout=10
% mysql -O connect_timeout=10
```

--set-variable 和-O 选项是不推荐的选项。

本附录在介绍各个程序时把与之有关的变量也列举出来。在程序的帮助信息里也能看到与该程序

有关的变量（参见 F.1 节）。

F.2.2 选项文件

大多数 MySQL 程序都支持选项文件的使用。作为一种保存程序选项的手段，选项文件使我们不必在执行程序时每次都在命令行上敲入一大堆的选项。你可以在 MySQL 安装目录找到选项文件，如果你有源代码发行版本，可以在 `support-files` 子目录里找到它们的名字为 `my-huge.cnf`、`my-large.cnf` 等（在 Windows 上，文件名后缀是`.ini`）。

如果你在命令行上设定的选项值与你事先在选项文件里设定的不一样，程序将使用前者进行有关的操作。

支持使用选项文件的 MySQL 程序将到好几个地方去寻找选项，但选项文件不存在却不会被视为出错。这意味着你通常都得自己创建选项文件。选项文件必须是文本文件，所以如果你在字处理器中创建选项文件，就一定要以普通文本格式保存它，而不能以字处理器的本地文档格式保存。

在 Unix 系统上，MySQL 程序将依次对表 F-2 中几个文件里的选项进行处理。

表 F-2

文 件 名	内 容
/etc/my.cnf	全局级选项
/etc/mysql/my.cnf	服务器级选项文件（从MySQL 5.1.15起）
SYSCONFDIR/my.cnf	全局级选项
$MYSQL_HOST/my.cnf	与特定服务器有关的选项
~/.my.cnf	与特定用户有关的选项

此外，如果文件是使用`--defaults-extra-file` 选项命名的，那么它会在`~/.my.cnf` 之前被读取，~表示主目录路径名。

`SYSCONFDIR` 来自构建 MySQL 时赋给 `configure` 的`--sysconfdir` 选项，其默认值是发行版本的安装目录下面的 etc 目录。这个选项文件位置从 MySQL 5.0.21/5.1.10 开始被采用，但在 5.0.53/5.1.22 之前，这个位置命名的文件一直是最后读取的。

`$MYSQL-HOME` 是一个环境变量，可以设置为包含特定于服务器的选项文件的目录，供 `mysqld_safe` 使用。如果没有设置它，`mysqld_safe` 将试图自动设置它，在 MySQL 安装目录或数据目录中查找 my.cnf 文件。

在 Windows 下，表 F-3 所示的选项文件是按顺序读取的。

表 F-3

文 件 名	内 容
WINDIR\my.ini、*WINDIR*\my.ini	全局选项
C:\my.ini、C:\my.cnf	全局选项
INSTALLDIR\my.ini、*INSTALLDIR*\My.ini	全局选项

此外，如果文件是通过`--defaults-extra-file` 选项命名的，那么它就会在其他文件之后被读取。*`WINDIR`* 是 Windows 目录的路径名（如 C:\Windows 或 C:WinNT）。*`INSTALLDIR`* 是 MySQL 安装目录的路径名。

只要是会用到选项文件的 MySQL 程序，就都要用到全局级选项文件。Unix 上用户级选项文件则

只有由该用户运行的程序才会用到。保存在服务器数据目录里的选项文件，只有那些以该目录作为默认数据目录的发行版本程序才会用到。数据目录现在是被弃用的选项文件位置。

Windows 用户在使用选项文件的时候还需要注意以下几个问题。

- Windows 路径通常包含有反斜线字符（\），可这个字符在 MySQL 里被当做转义字符使用。因此，你必须把取值为路径名的选项里的反斜线字符写成斜线字符（/）或双写的反斜线字符（\\）。
- Windows 往往会把文件名中的扩展名隐藏起来。如果你创建了一个名为 my.cnf 的选项文件，Windows 可能会把它显示为 my。如果你发现了这个“错误”并在重新命名为 my.cnf，就会发现它不再起作用了，因为你刚才把它的名字从 my.cnf 改成 my.cnf.cnf 了！

有几个与选项文件的处理工作有关的选项是大多数 MySQL 程序都能识别和支持的，下面列出的是它们的含义。只要你打算使用它们当中的任何一个，就必须把它写成命令行上的第一个选项。

- `--defaults-extra-file = `*`file_name`*

  除标准的选项文件外，MySQL 程序还必须从这个文件里读取选项。程序将在读完全局级和服务器级选项文件之后、在读取用户级选项文件之前去读取这个文件。从 MySQL5.0.6 版本开始，这个文件必须存在，并且可读，否则将出错。
- `--defaults-file = `*`file_name`*

  只从这个文件里读取选项。在默认的情况下，程序会依次到好几个地方去寻找选项文件，但如果你给出了`--default-file` 选项，它将只读取指定文件。这个必须存在且可读，否则将出错。
- `--defaults-group-suffix=`*`suffix`*

  读取具有默认名字的选项组，以及名字为默认值加上指定后缀的选项组。这个选项是从 MySQL 5.0.10 开始引入的。
- `--no-defaults`

  禁止使用任何选项文件。此外，这个选项还会导致`--defaults-file` 等与选项文件有关的其他选项失去作用。
- `--print-defaults`

  因为有选项文件和环境变量的缘故，所以即使你没有在命令行上给出任何选项，MySQL 程序也会按照一些“默认的”选项设置去执行。`--print-defaults` 可以用来检查某个选项文件的设置情况是否正确。此外，如果 MySQL 程序的行为看起来像是使用了一个你从没给出过的选项，你就应该使用`--print-defaults` 选项来检查一下，看它是不是来自某个选项文件。

你可以在帮助信息中看到程序通常读取的选项文件的清单。参见 F.1 节。这份清单会受到`--defaults-file`、`--defaults-extra-file`、`--no-defaults` 等选项的影响。

选项是分组给出的。下面是一个示例：

```
[client]
user=sampadm
password=secret

[mysql]
skip-auto-rehash

[mysqlshow]
status
```

选项组的名字必须写在方括号里，不区分大小写。`[client]`是一个特殊的组名，这一组里的选项将作用于所有的客户程序。其他的组名通常都对应于某个具体的客户程序。在上面的例子里，组名`[mysql]`

表示该选项是供 mysql 客户程序使用的，组名[mysqlshow]则表示该组选项是供 mysqlshow 客户程序使用的。标准的 MySQL 客户程序会到[client]选项组和与它同名的选项组里去查找选项。比如说，mysql 会到[client]和[mysql]选项组里查找，mysqlshow 则会到[client]和[mysqlshow]里去查找选项。

注意不要把选项放在只有一个客户能理解的[client]组里。例如，skip-auto-rehash 特定于 mysql。如果把选项放在[client]组，你会发现 mysqlimport 等其他客户程序不再工作了。(帮助信息后会出现一条出错报告。) 所以应把 skip-auto-rehash 放在[mysql]组里。

跟在某个组名后面的选项都与该组相关联。一个选项文件可以包含任意多个组，后出现的组将优先于先出现的组。如果某个选项在程序查找的多个选项组里出现，该程序最终将使用这个选项最后一次出现时的值。

在选项文件里，每个选项都独占一行。每行的第一个单词是该选项的名字，它必须以不带前导连字符的长格式形式写出。(比如说，在命令行上，可以用-C 或--compress 设置压缩功能；但在选项文件里，只能使用 compress。) 只要是程序支持的长格式选项，就可以写到选项文件里。选项和选项值（如果有的话）要用等号（=）隔开。

请看下面这条命令行命令：

```
% mysql --compress --user=sampadm --max_allowed_packet=16M
```

如果想用选项文件里的[mysql]组来给出同样的设置信息，可以这样做：

```
[mysql]
compress
user=sampadm
max_allowed_packet=16M
```

你可以用一个引号或双引号来括住一个选项值，如果值包含空白的话，这种做法是有用的。

选项文件里的空白行、以“#”或“;”开始的行将被视为注释而不做处理。选项设置行上的前导空格（如果有的话）将被忽略。你可以在行中用“#”(但不能用“;”) 字符开始一条注释。

在选项文件里，某些特殊字符需要用转义序列来表示（如表 F-4 所示）。

表 F-4

转义序列	含　义
\b	退格符
\n	换行符
\r	回车符
\s	空格
\t	制表符
\\	反斜线字符

从 MySQL 5.0.4 起，选项文件可以包含使其他选项文件被读取的命令。

- !include *file_name*

  读取指定选项文件。
- !includedir *dir_name*

  读取指定目录中的所有选项文件。指定的是 Unix 上有扩展名.cnf 的选项文件或者 Windows 上有扩展名.ini 或.cnf 的选项文件，读取文件的顺序不一定。

包含的文件跟在默认选项文件语法后面，只有当前包含的选项组中的选项才会被使用。

1. 让用户级选项文件做到专人专用

在 Unix 系统上，先设置好文件的属主，再把文件的访问模式设置为 600 或 400，其他用户就不能读出它的内容了。利用这一点，我们就能让用户级选项文件做到专人专用，谁都不想让自己的 MySQL 用户名和口令信息被其他用户偷看到。如果想让你自己的选项文件只能由你本人来读取，就需要在登录子目录里发出下面这样的命令：

```
% chmod 600 .my.cnf
% chmod go-rwx .my.cnf
```

2. 用my_print_defaults来查看选项

如果你想知道程序都使用了哪些选项，可以用 my-print_defaults 工具来查看。这个工具将从选项文件里把与该程序有关的选项都查找并显示出来。比如说，mysql 程序要使用来自[client]和[mysql]组的选项。如果想知道选项文件里都有哪些会应用到 mysql 程序的选项，就要调用 my_print_defaults。

```
% my_print_defaults client mysql
```

再比如说，服务器程序 mysqld 要使用来自[mysqld]和[server]组的选项。如果想知道这两个选项组都列举了哪些选项，就要使用下面这样的命令：

```
% my_print_defaults mysqld server
```

F.2.3　环境变量

MySQL 程序会检查一些环境变量以获得选项设置。环境变量的优先程度很低，在选项文件里或者在命令行上设置的选项都优先于用环境变量设置的选项。

MySQL 程序检查以下环境变量。

- MYSQL_DEBUG

  调试时使用的选项。如果在编译 MySQL 软件时没有启用调试支持机制，这个变量将没有任何效果。设置 MYSQL_DEBUG 变量与使用--debug 选项的情况相同。

- MYSQL_PWD

  用来连接 MySQL 服务器的口令。设置 MYSQL_PWD 变量与使用--password 选项的情况相同。用 MYSQL_PWD 变量来保存口令是不安全的，你系统上的其他用户可以轻易发现它的值。比如说，ps 命令能把其他用户的环境变量设置显示出来。

- MYSQL_TCP_PORT

  对于客户程序，这是它以 TRCP/IP 方式连接服务器时使用的端口号。对于 mysqld，这个选项指定的是它将在其上监听 TCP/IP 连接的端口。设置 MYSQL_TCP_PORT 变量与使用--port 选项的情况相同。

- MYSQL_UNIX_PORT

  对于客户程序，这是它在连接主机 localhost 上的服务器时使用的套接字文件的路径名。对于 mysqld，这是它在其上监听本地连接的套接字。设置 MYSQL_UNIX_PORT 变量与使用--socket 选项的情况相同。

- TMPDIR

  一个路径名，MySQL 将把临时文件创建在这个子目录里。设置这个变量与使用--tmpdir 选项的情况相同。然而，即使 myisamchk 和 mysqld 理解包含--tmpdir 目录的值，也不要那样

设置 TMPDIR。不理解目录列表的其他非 MySQL 程序也会使用 TMPDIR。

- USER

  用来连接服务器的 MySQL 用户名。这个选项只能由运行在 Windows 或 NetWare 系统下的客户程序使用，设置这个变量与使用--user 选项的情况相同。

mysql 客户程序还要多检查 3 个环境变量。

- MYSQL_HISTFILE

  在 Unix 上，这是存储交互使用中命令行历史时用到的文件名。这个变量的默认值是 $HOME/.mysql_history，$HOME 是登录子目录的位置。

- MYSQL_HOST

  将要连接的主机（即 MySQL 服务器在其上运行的主机）。设置这个变量与使用--host 选项的情况相同。

- MYSQL_PS1

  代替 mysql>充当主提示符的字符串。这个字符串可能会包含有我们将在 F.5 节介绍的特殊序列。

F.3　myisamchk

这个工具程序能够完成以下工作：检查和修复损坏的数据表，显示数据表信息，对索引键值的分布情况进行分析，禁用或启用索引。第 5 章对键值分析和索引禁用等问题做了比较详细的介绍。第 14 章对数据表的检查和修复工作做了比较详细的介绍。

myisamchk 用来对 MyISAM 存储格式的数据表进行处理，这类数据表的数据和索引文件分别以.MYD 和.MYI 为文件扩展名。如果你用 myisamchk 去处理 ISAM 数据表，它将显示一条警告信息并忽略该数据表。

使用 myisamchk 对某个数据表进行检查：

```
myisamchk [options] tbl_name[.MYI] ...
```

如果没有给出任何选项，这个程序将去检查给定数据表是否有错误。如果给出了一些选项，它们就将按照那些选项的含义去进行操作。如果打算进行的操作会改变数据表的内容，就应该先备份数据表。

每个 *tbl_name* 参数可以是一个数据表的名字，也可以是该数据表的索引文件的的名字，以.MYI 为扩展名。使用索引文件名的好处是你可以利用文件名通配符只用一条命令就对多个数据表进行了处理。比如说，如果想对当前目录里所有的 MyISAM 数据表进行检查，只需发出下面这样的命令就行了：

```
% myisamchk *.MYI
```

这个程序并不要求数据表必须存放在某个地方。如果想检查的数据表不在当前子目录里，就必须给出它们所在的子目录的路径名。因为这个程序不要求数据表文件必须存放在服务器的数据目录里，所以你可以先把它们复制到另外一个子目录，然后再对那些副本文件而不是原始文件进行操作。

myisamchk 执行的许多操作也可以通过 SQL 语句来完成，如 ANALYZE TABLE、CHECK TABLE、OPTIMIZE TABLE 和 REPAIR TABLE。你可以直接发出这些语句，也可以使用 mysqlcheck 程序，这个程序为几个维护表的语句提供了命令行接口。一般而言，使用这些语句或 mysqlcheck 比使用 myisamchk 更简单和安全。

使用 myisamchk 在表上进行维护时必须防止服务器同时访问表文件，否则会毁坏数据表。如果你真的想使用 myisamchk，请阅读 14.1 节，其中讨论了如何进行这种阻止。

在下面两个条件都具备的情况下，如果在包含 FULLTEXT 索引的表上使用 myisamchk，也要特别小心。

- 使用 myisamchk 执行修改索引的操作，包括分析和修复操作。
- 使用 FULLTEXT 相关的任何系统变量(ft_max_word_len、ft_min_word_len 或 ft_stopword_file）的非默认值运行服务器。

在上面两个条件都具备的情况下，必须使用合适的选项来告诉 myisamchk，应该使用哪个 FULLTEXT 参数，因为它不知道服务器在使用哪个值。如果你不这么做，myisamchk 构建 FULLTEXT 索引时使用的参数值将不是服务器期望的，而且 FULLTEXT 搜索将返回错误结果。假设你使用 ft_min_word_len 和 ft_stopword_file 的以下非默认选项设置来运行服务器：

```
[mysqld]
ft_min_word_len=2
ft_stopword_file=/var/mysql/data/my-stopwords
```

在这种情况下，对于你在包含 FULLTEXT 索引的表上执行的任何索引修改操作，你都必须将这些值也指定给 myisamchk，你可以在命令行上使用-ft_min_word_len 和--ft_stopword_list 选项来完成，但最好将值记录到选项文件里，这样你就不会忘记使用它们。使用选项组的情况类似于在服务器上的情况：

```
[myisamchk]
ft_min_word_len=2
ft_stopword_file=/var/mysql/data/my-stopwords
```

通过使用 REPAIR TABLE 或 ANYLYZE 等表维护语句可以完全避免 FULLTEXT 参数错误匹配的问题。然后服务器进行索引修改操作，又因为它知道使用哪些 FULLTEXT 参数，所以它会将它们用于维护包含 FULLTEXT 索引的数据表。

F.3.1　**myisamchk**支持的标准选项

```
--character-sets-dir  --set-variable        --version
--debug               --silent
--help                --verbose
```

--silent 选项意味着只显示出错信息，而--verbose 选项在你同时给出了--check 、--description 或--extend-check 选项的时候将显示更多的信息。在同一条命令里给出多个--silent 或--verbose 选项将加强它们的效果。

F.3.2　**myisamchk**支持的选项

有些选项需要用到索引的序号。索引是从 1 开始编号的。你可以用 SHOW INDEX 查询或者 mysqlshow --keys 命令来查看特定数据表各个索引的编号顺序，而 myisamchk 将按照各索引在 Key_name 输出列里的先后顺序对它们进行检查和修复。

- --analyze 或 -a

  进行键值分布分析。这可以帮助服务器加快基于索引的查找和联结操作的执行速度。完成分析工作之后，以--description 加--verbose 选项再次运行 myisamchk 程序，你就能查看到键值分布信息了。

- --backup 或 -B

  在其他选项会对数据文件（.MYD）作出修改的情况下，这个选项将先对数据表进行备份，备份文件的名字是 tbl_name-time.BAK。time 是一个时间戳的数值表示形式。备份文件将被写

到原始数据表所在的子目录里。

- ❑ --block-search=*n* 或 -b *n*

  数据表的第 *n* 个区块是从第几个数据行开始的。这个选项仅用于调试工作。
- ❑ --check 或 -c

  检查数据表中的错误。这是在你没有给出任何选项时的默认动作。
- ❑ --check-only-changed 或-C

  只对上次检查后又发生过修改的数据表进行检查。
- ❑ --correct-checksum

  对于使用了 CHECKSUM = 1 选项而创建出来的数据表，这个选项将确保该数据表里的校验和信息正确无误。
- ❑ --data-file-length=*n* 或 -D *n*

  当数据文件的长度增长到 MySQL 软件本身或者操作系统所容许的上限，或者当数据表里的数据行的个数增长到 MySQL 内部数据结构所容许的上限时，我们将无法再往里面添加新的数据记录。此时，我们需要重建这个数据文件并设定一个最大长度。这个选项的设置值以字节为计量单位。这个选项必须与--recover 或--safe-recover 选项一起使用才会生效。
- ❑ --description 或 -d

  显示关于数据表的描述性信息。
- ❑ --extend-check 或 -e

  对数据表做进一步的检查。需要用到这个选项的场合是非常少的，因为 myisamchk 已经足以把所有的错误都查出来了。
- ❑ --fast 或-F

  检查没有正确关闭的数据表。例如，如果 mysqld 在打开表的同时服务器主机崩溃了，就需要这个选项，这样，mysqld 就没有机会关闭它们。
- ❑ --force 或 -f

  强制进行对数据表的检查或修复工作，即使已经存在一个对应于该数据表的临时文件。一般说来，如果 myisamchk 发现已经存在一个名为 tbl_name.TMD 的文件，它就会在显示一条出错信息后退出执行，因为这通常意味着有另外一个 myisamchk 程序实例正在运行。但也存在这样一种可能性：你在程序尚未执行完毕时强行终止了它——因为来不及把临时文件安全地删除掉，所以它们就遗留在了系统里。如果你知道发生过这样的事情，就应该用--force 选项来执行程序，不管是否存在临时文件都强行运行。（另一种做法是手动删除临时文件。）

  如果你在检查数据表时使用了--force 选项，那么，只要 myisamchk 发现某个数据表有问题，就会自动使用--recover 选项重新启动。此外，myisamchk 还将自动刷新数据表的状态，就好像你还同时使用了--update-state 选项那样。
- ❑ --information 或 -i

  显示关于数据表内容的统计信息。
- ❑ --keys-used=*n* 或 -k *n*

  这个选项要与--recover 选项配合使用。*n* 将被用作一个表明允许使用哪些索引的位掩码，第一个索引对应于第 0 位。（例如，值为 6 时表示二进制数 110，表明应该使用第二个和第三个个索引。）把 *n* 设置为 0 意味着要禁用所有的索引。如果使用得当，这个选项将改善 INSERT、DELETE、UPDATE 等操作的性能。重新启用某个索引将恢复其正常的索引行为，把掩码 *n* 中与

各有关索引相对应于的比特位全都设置为 1 即可。

- --max-record-length = *n*
  如果不能为它们分配内存，就忽略大于 *n* 字节的行。
- --medium-check 或 -m
  对数据表进行中级修复。它比--extend-check 方法执行得快，但不如后者那么彻底(myisamchk 程序的帮助信息说这个方法“只能找出 99.99%的错误”。)这种检查模式对大多数场合来说都应该足够了。中级检查模式的工作原理是这样的：先把索引中的键值的 CRC 校验和计算出来，再把它们与根据数据文件中的被索引数据列计算出来的 CRC 校验和进行比较。
- --parallel-recover 或 -p
  像--recover 选项那样修复数据表，但将使用多个线程并行地重建索引。这要比以非并行方式重建索引的速度更快，但这个选项目前仍被认为是试验性质的。
- --quick 或 -q （布尔）
  与--recover 选项配合使用，与单独使用--recover 选项更快。给--recover 选项加上--quick 选项将不对数据文件进行修复。如果想在发现重复键值时对数据文件进行修复，就需要给--recover 选项加上两个--quick 选项。
- --read-only 或-T
  不将数据表标记为已检查。
- --recover 或 -r
  对数据表进行常规的修复操作。这可以修复数据表的大多数错误，但不能消除唯一化索引中的键值重复现象。
- --safe-recover 或 -o
  在确保安全的前提下对数据表进行修复。虽说比--recover 方法执行得慢，但这个选项能够修复一些--recover 选项不能修复的错误。此外，--safe-recover 使用的磁盘空间也要少于--recover。
- --set-auto-increment[=*n*] 或 -A[*n*]
  对 AUTO_INCREMENT 计数器进行设置，此后的序列编号值将从 *n* 开始，如果数据表里已经存在有序列编号等于 *n* 的记录，将从比 *n* 更大的值开始。如果没有给出 *n* 值，这个选项将把下一个 AUTO_INCREMENT 值设置为“当前最大编号值+ AUTO_INCREMENT 递增量”。
  如果使用的是这个选项的短格式，-A 和 *n* 值之间将不允许有空格；如果有空格的话，MySQL 就将无法对 *n* 值作出正确的解释。
  你可以不使用 myisamchk 为 MyISAM 表设置 AUTO_INCREMENT 值，只需发出下面的语句：
  ```
  ALTER TABLE tbl_name AUTO_INCREMENT = n;
  ```
- --set-character-set=*charset*
  在重建索引的时候，使用给定字符集的排位顺序来确定各索引数据项的顺序。这个选项已从 MySQL5.0.3 中移除，被--set-collation 代替。
- --set-collation=*collation*
  重建和排序表的索引项时用到的排序方式，名称表示字符集名称。这个选项从 MySQL 5.0.3 开始引入，替代了--set-character-set。
- --sort-index 或 -S
  对索引区块进行排序以加快此后检索操作读取多个索引区块的速度。

- --sort-records=*n* 或 -R *n*
  根据数据记录在第 *n* 个索引中的先后顺序对它们排序。这将加快基于该索引的检索操作的执行速度。在你第一次对数据表进行这种操作时，因为数据记录从没进行过排序，所以它的速度可能会很慢。可以用 ALTER TABLE ... ORDER BY 语句来完成与--sort-records 选项同样的工作，而且在速度上往往会更快一些。
- --sort-recover 或 -n
  对数据表强制进行排序模式恢复，不管进行这种恢复所必需的临时文件会增大到什么程度。
- --start-check-pos=*n*
  从位置 *n* 开始读取数据文件。这个选项只用在调试工作中。
- --tmpdir=*dir_name* 或 -t *dir_name*
  用来存放临时文件的子目录的路径名。这个选项的默认值是环境变量 TMPDIR 的取值；如果你没有设定这个环境变量，则以/tmp 为默认值。这个选项的值可以是一组将以轮转方式使用的子目录。在 Unix 系统上，子目录名之间要用冒号（:）分隔；在 Windows 或 NetWare 系统上，子目录名之间要用分号（;）分隔。
- --unpack 或 -u
  对用 myisampack 程序压缩的文件进行解压缩。这个选项可以用来把压缩的只读数据表转换为可修改的格式。它不能与--quick 或--sort-records 选项一起使用。
- --update-state 或 -U
  对保存在数据表内部的状态标志进行刷新。完好的数据表将被标志为一切正常，有缺陷的数据表将被标记为需要修复。这个选项将使今后以--check-only-changed 选项执行的 myisamchk 程序在完好的数据表上执行得更有效率。
- --wait 或 -w
  如果数据表已被锁定，则等待到该数据表可用为止。如果没有使用-wait 和 myisamchk 选项，在遇到被锁定的数据表时将等待 10 秒，然后——如果到那时还没有获得数据锁的话——显示一条出错信息。

F.3.3 与**myisamchk**有关的变量

下面这些与 myisamchk 有关的变量都可以按 F.2.1 节中第 2 小节所介绍的步骤进行设置。

对于包含 FULLTEXT 索引的表，注意 myisamchk 程序描述中的注意事项。

- decode_bits
  在对压缩数据表进行解码时使用的二进制位的个数。这个值越大，解码操作也就越快，但会消耗较多的内存。一般说来，这个变量的默认值 9 已经足以应付大多数情况了。
- ft_max_word_len
  允许包括在 FULLTEXT 索引里的单词的最大长度，长于这个长度的单词将被忽略。默认值为 84。
- ft_min_word_len
  允许包括在 FULLTEXT 索引里的单词的最小长度，短于这个长度的单词将被忽略。默认值为 4。
- t_stopword_file
- FULLTEXT 索引的 stopword_file，没有默认值，这意味着“使用内建的 stopword”。
- key_buffer_size
  用来存放索引区块的缓存区的长度。这用于--safe-recover，但不用于--recover 和

--sort-recover。默认值是 512KB。

- key_cache_block_size
  键缓冲区中块的大小，默认值是 1 MB。
- myisam_block_size
  MYI 文件中索引块的块大小。默认值为 1 MB。
- read_buffer_size
  读操作的缓冲区的长度。默认值为 256 KB。
- sort_buffer_size
  用来完成索引键值排序操作的缓冲区的长度。（--recover 操作会用到这个缓存区，但 --safe-recover 操作不会用到它。）默认值为 2 MB。
- sort_key_blocks
  这个变量与数据表的索引所使用的二元树数据结构的深度有关。这个值默认为 16，很少需要修改。
- stats_method
  在搜集索引键值分布统计信息时，NULL 值是否应视为相等。如果这个变量为 null_equal，则表示所有 NULL 值形成一个组；如果为 null_unequal，则表示每个 NULL 值形成一个单独的组。这个变量是从 MySQL 5.0.14 引入的，此前的统计信息计算类似于 null_equal 方法。
- write_buffer_size
  写操作的缓冲区的长度。默认值为 256 KB。

F.4 **myisampack**

myisampack 工具程序将生成压缩的只读数据表。在确保你仍能快速访问数据记录的同时，它们在一般情况下能减少大约 40%~70%的存储空间占用量。myisampack 程序负责压缩 MyISAM 数据表，能够对所有的数据列类型进行压缩。

MySQL 软件的任何版本都能把用这个工具程序生成的压缩数据表的内容读出来。因此，如果你的应用软件所涉及的数据表只包含只读信息而不需要修改——比如档案或者百科全书之类的东西，就很值得在发行软件之前先用这个工具程序来压缩数据表。比如说，如果应用软件使用了嵌入式服务器且将以 CD-ROM 光盘的形式发行，对 MyISAM 数据表进行压缩就将使你能够把更多的数据保存在光盘上。

运行 myisampack 程序并列出打算压缩的数据表的名字：

```
myisampack [options] tbl_name ...
```

每个 *tbl_name* 参数可以是一个数据表的名字，也可以是该数据表的索引文件的的名字。（MyISAM 数据表的索引文件以.MYI 为扩展名。）如果想压缩的数据表不在当前目录里，就必须把它们所在的子目录的路径名也包括在 *tbl_name* 参数里。

myisampack 程序只压缩数据文件，对索引文件没有影响。因此，在运行完 myisampack 程序后，还必须用 myisampack --recover --quick 命令刷新索引。

如果想把压缩文件转换为允许修改的非压缩格式，可以使用 myisampack --unpack 命令。

F.4.1 **myisampack**程序支持的标准选项

```
--character-sets-dir  --help        --verbose
--debug               --silent      --version
```

F.4.2 **myisampack**程序独有的选项

- --backup 或 -b
  在对每一个 *tbl_name* 参数所给定的数据文件进行压缩之前，先制作一个备份。备份文件的名字将是 *tbl_name*.OLD。
- --force 或 -f
  对数据表强制压缩，不管压缩结果文件的长度是否大于原始文件的长度，也不管是否已经存在该数据表的临时文件。一般说来，如果 myisampack 程序发现已经存在名为 *tbl_name*.TMD 的文件，它就会在显示一条出错信息后退出执行，因为这通常意味着有另外一个 myisampack 程序实例正在运行。但也存在着这样一种可能性：你在这个工具程序尚未执行完毕的时候强行终止了它——因为来不及把临时文件安全地删除掉，所以它们就遗留在了系统里。如果你知道自己的系统曾发生过这样的事情，就应该用--force 选项来执行这个工具程序，让它不管是否存在临时文件都强行运行。（另一种做法是手动删除临时文件。）
- --join=*join_tbl* 或 -j *join_tbl*
  把你在命令行上给出的所有数据表合并为一个名为 *join_tbl* 的压缩数据表。被合并的数据表必须具有同样的结构（即数据列的名字、类型、索引都必须一模一样）。这个选项与 MERGE 表无关。这个操作不会为输出数据表创建一个.frm 文件，但你可以在 myisampack 程序执行完毕后从一个源数据表复制.frm 文件自行创建一个。
- --test 或 -t
  让 myisampack 程序在测试模式下运行。该程序将“假装”进行压缩，并把你在真正压缩时会看到的各种信息显示出来。
- --tmpdir=*dir_name* 或 -T *dir_name*
  用来存放临时文件的子目录的路径名。
- --wait 或 -w（布尔）
  如果打算压缩的数据表正被其他客户程序使用，则等待并重试。（如果某个数据表在压缩的过程中可能会被修改，就不应该压缩它。）

F.5 **mysql**

mysql 客户程序可以用来连接服务器、发出 SQL 语句、查看查询结果。

```
mysql [options] [db_name]
```

如果给出了一个 *db_name* 参数，该数据库将成为本次会话期间的默认数据库。否则，mysql 程序将以没有默认数据库的方式启动，而你则必须在今后的查询命令里以 *db_name*.*tbl_name* 的形式来引用每个数据表，或是先用一条 USE *db_name* 命令来指定一个默认数据库。

mysql 程序可以交互运行。你也可以在批处理模式下使用 mysql 程序来执行保存在某个文件里的查询命令，只需像下面这样把它的输入重定向为那个文件即可：

```
% mysql -u sampadm -p -h cobra.snake.net sampdb < my_sql_file
```

在交互模式下，mysql 程序会在启动后显示一个 mysql>提示符以表明它正在等待输入。发出一条语句的办法是这样的：在 mysql>提示符处敲入该语句的文本（如有必要，输入的内容允许跨越多行），再敲入一个分号（;）或“\g”作为语句的结束标志。mysql 程序将把该语句发送到服务器去执

行，显示查询结果，然后再次显示 mysql>提示符以表明它在等待输入下一条语句。“\G”也可以用作语句的结束标志，但它将纵向显示查询结果（即每个数据列值占据一行）。

你敲入输入内容时，mysql 程序会改变提示符以表明它正在等待的内容，如表 F-5 所示。mysql>提示符是主提示符，其含义是“请输入一条新的语句”。其他提示符都属于辅助提示符，表示还得再输入一些内容才能完成当前的查询命令。

表　F-5

提　示　符	含　　义
mysql>	等待用户输入一条新语句的第一行
->	等待用户输入当前语句的下一行
'>	等待用户为当前语句敲入一个配对的单引号
">	等待用户为当前语句敲入一个配对的双引号
`>	等待用户为当前语句敲入一个配对的反引号
/*>	等待用户敲入一个配对的“*/”注释记号

“'>”和“">”提示符表明你在前一行敲入了一个单引号或双引号，但尚未输入与之配对的结束引号。类似地，“`>”提示符表明还需要敲入一个配对的反引号。“/*>”提示符表明用“/*”记号开始了一个“/*…*/”注释，但还没有敲入配对的“*/”记号。一般来说，如果你看到了这些提示符，那十有八九是因为你忘记结束一个字符串、标识符或注释。在遇到这种情况时，如果想立刻退出这类字符串收集模式，请先根据提示符敲入一个配对的引号或注释结束标记，再敲入“\c”以取消当前语句。

在 Unix 系统上，当 mysql 程序在交互模式运行时，它会把你敲入的查询命令保存到一个历史文件里去。这个文件的默认路径名是$HOME/.mysql_history，但你可以通过环境变量 MYSQL_HISTFILE 来改变它。

有些选项会抑制这种历史记录功能。一般来说，那些将导致 mysql 程序以非交互方式运行的选项（如--batch、--html、--qiuck 等）都有这样的效果。

在支持 Readline 库的系统上，我们可以把我们以前输入过的语句从命令历史里调出来再次执行，在执行之前还可以先编辑。在 Windows 系统上，因为它不支持 Readline 库，所以我们将无法使用 Readline 库提供的编辑功能，但 Windows 内部支持的几种编辑命令都可以在 mysql 程序里使用。如果想了解如何使用 Readline 库或 Windows 操作系统提供的命令行编辑功能，请参阅 1.5.2 节的第 1 小节。

F.5.1　**mysql**程序支持的标准选项

```
--character-sets-dir       --host                     --silent
--compress                 --password                 --socket
--debug                    --pipe                     --user
--debug-check              --port                     --verbose
--debug-info               --protocol                 --version
--default-character-set    --set-variable
--help                     --shared-memory-base-name
```

--debug-check 选项从 MySQL 5.1.21 版本开始就可以使用了。

--debug-info 从 MySQL5.1.14 开始成为标准选项，在此前，它也显示查询结果元数据，从 5.1.14 开始使用—column-type-info 显示元数据。

mysql 程序还支持各种标准的 SSL 选项。

在同一条命令里给出多个--silent 或--verbose 选项将加强它们的效果。

-I 是-help 的同义词。

F.5.2 mysql独有的选项

- --auto-rehash（布尔）

  在启动时，mysql 程序会对数据库、数据表、数据列的名字进行散列计算并构造出一个用来实现名字自动补足功能的数据结构。此后，当输入查询命令时，你只需输入某个名字的前几个字符再按下 Tab 键，mysql 程序就会（如果不会产生二义性的话）自动补足这个名字。

  在默认情况下，这种散列计算的机制是处于启用状态的，但如果你没有选择默认数据库，它就不会起作用。--skip-auto-hash 选项将禁用这一机制，使 mysql 程序能够更快地启动，尤其是在你有很多数据表的场合。

  如果你在启动 mysql 程序时禁用了散列计算的机制，可稍后又想使用名字自动补足功能，可以在 mysql>提示符处使用 rehash 命令。
- --auto-vertical-output（布尔）

  为超出终止宽度的查询结果采用自动的垂直输出样式。这个选项从 MySQL 6.0.4 开始引入。
- --batch 或 -B

  以批处理方式运行 mysql 程序。查询结果将显示为制表符间隔格式（每个数据行占据一个输出行，各数据列值之间用制表符分隔）。这种格式的输出报告非常适合被导入到其他程序（比如电子试算表软件）做进一步的处理。在默认情况下，查询结果输出报告的第一行将是一个由各数据列名称组成的标题行。如果不想看到这行标题，就需要使用--skip-column-names 选项。
- --column-names（布尔）

  在查询结果的输出报告里显示数据列的名字，若不想显示，则使用--skip-column-names。通过使用两次--silent 也可以达到这种效果。
- --clolumn-type-info 或-m（布尔）

  在查询输出中包含结果集元数据。这个选项从 MySQL5.1.14（-m 是从 MySQL5.1.21）开始引入的。此前使用--debug-info 获取结果集元数据。
- --comments 或-c（布尔）

  如果语句包含注释，则语句发送到服务器时也包含注释。注释是默认截掉的，就好像指定了--skip-comment。这个选项是从 MySQL5.0.52/5.1.23 开始引入的。
- --database=*db_name* 或 -D *db_name*

  设定默认数据库。
- --delimiter=*str*

  设定语句分隔符，默认为“;”。
- --execute=*stmt* 或 -e *stmt*

  执行查询命令并退出。必须把查询命令放在引号里以避免你的 shell 把该查询命令解释为多个命令行参数。可以一次给出多个查询命令，但必须用分号把它们彼此分隔开。
- --force 或 -f（布尔）

  在从一个文件读取查询命令时，如果发生错误，mysql 程序通常会退出执行。如果你给出了这

个选项，mysql 将忽略这种出错并继续处理查询命令。

- --html 或 -H（布尔）
  生成 HTML 输出。
- --i-am-a-dummy（布尔）
  这个选项是--safe-updates 的同义词。
- --ignore-spaces 或 -i
  让服务器忽略内建函数名与引导其输入参数表的左括号字符“(”之间的空格。在默认情况下，左括号字符必须紧跟在函数名的后面，它们之间不允许有间隔。这个选项将使函数名被当做保留字来对待。
- --line-numbers（布尔）
  在出错信息里显示行号，这是默认行为；如果不想看到行号，就需要使用--skip-line-numbers 选项。
- --local-infile（布尔）
  启用或者禁用 LOAD DATA LOCAL 语句。LOCAL 功能虽然已经出现，但默认设置是处于禁用状态的。如果发出的 LOAD DATA LOCAL 语句导致了一条出错信息，请在以--local-infile 选项重新启动 mysql 程序后再试一次。这个选项还可以用来关闭 LOCAL 功能（如果它正处于启用状态的话），例如，使用--disable-local-infile 选项。
  如果服务器被配置成不允许使用 LOCAL 功能，这个选项将不会有任何效果。
- --named-commands 或 -G （布尔）
  允许 mysql 程序的内部命令的长格式形式出现在任何输入行的开头部分。如果用--skip-named-commands 选项禁用了这一功能，长格式命令就只允许出现在主提示符处而不允许出现在辅提示符处，也就是说，不允许出现在多行语句的第二及后续各行上。
- --no-auto-rehash 或 -A
  请参见对-auto-rehash 选项的介绍。--no-auto-rehash 已经被淘汰，支持采用--skip-auto-rehash。
- --no-beep 或 -b（布尔）
  在发生错误时不发出蜂鸣报警音。
- --no-named-commands 或 -g
  请参见对--named-commands 选项的介绍。--no-named-commands 已经被淘汰。
- --no-pager
  请参见对--pager 选项的介绍。这个选项已被--skip-pager 所取代。
- --no-tee
  请参见对--tee 选项的介绍。这个选项已被--skip-tee 所取代。
- --one-database 或-o
  这个选项用在需要根据某个二进制日志文件的内容修改数据库的场合。它告诉 mysql 程序只对默认数据库（即在命令行上指定的数据库）做出修改，对其他数据库的修改将被忽略。如果没在命令行上指定数据库，则不进行任何修改。
- --pager[=*program*]
  使用分页显示程序（比如/bin/more 或/bin/less）来分页显示比较长的查询结果，每次显示一页。如果没有给出 *program* 参数，将根据环境变量 PAGER 的值来决定将使用哪一个分页显示

程序。分页显示功能在批处理模式下不可用，在 Windows 系统上也不可用。可以用--disable-pager 选项来禁用这项功能。

- --prompt=*str*

  把 mysql 程序的主提示符从 mysql>改变为由 str 定义的字符串。这个字符串可以包含特殊的序列，具体情况见 F.5.5 节。

- --quick 或 -q

  在默认情况下，mysql 程序从服务器把查询结果集全部检索出来之后再显示。这个选项将使程序每检索出一个数据行就立刻显示，这既减少了内存的占用量，又能使那些不这样做就会失败的大查询得以成功地完成。不过，最好不要在交互模式下使用这个选项，因为如果用户暂停了输出或者挂起了 mysql 程序，服务器就会进入等待状态，进而影响到其他客户操作。

- --raw 或 -r（布尔）

  把数据列值原封不动地显示出来，不对其中的特殊字符做任何转义处理。这个选项通常要与--batch 选项联合使用。

- --reconnect（布尔）

  如果连接中断，自动重新连接服务器。在 MySQL 5.0.3 之前，这个选项是默认启用的。若要禁用，采用--skip-reconnect。

  自动重新连接有时候会招来麻烦。例如，任何当前的活动事务都会回滚，会话变量会丢失，而且毫无提示。

- --safe-updates 或 -U（布尔）

  这个选项给数据库的修改操作加上了一些限制，这些限制对 MySQL 新手很有好处。

  - 如果启用了这个选项，MySQL 将只允许会改变数据的语句在以下两种情况下执行：（1）将被修改的记录是通过键值确定的；（2）使用了 LIMIT 子句。这有助于防止查询命令意外地改变或者删除数据表的全部或者大部分内容。
  - 这个选项还将把非联结检索和联结检索的结果集分别限制在 1 千（除非采用了 LIMIT 子句）和 1 百万个数据行以下。这有利于防止意外生成过大的查询结果。

通过设置 select_limit 和 max_join_size variables 可以改变这些限制。

- --secure-auth（布尔）

  禁止与服务器连接，除非它支持 MySQL 4.1 中引入的更安全的口令格式。

- --show-warnings（布尔）

  为每个语句自动显示警告。这个选项是从 MySQL 5.0.6 开始引入的。

- --sigint-ignore（布尔）

  忽略 SIGINT 信号。它通常通过输入 Ctrl-C 来发送，在 MySQL 5.0.25 之前，这会导致 mysql 退出。从 MySQL 5.0.25 起，Ctrl-C 会让 mysql 删除当前语句。如果没有删除语句或者在删除前输入了其他 Ctrl-C，mysql 将会退出。使用--sigint-ignore 可以让 mysql 像刚才描述的那样解释 Ctrl-C。这个选项是从 MySQL 5.0.2 开始引入的。

- --skip-column-names 或 -N

  -N 形式已经被淘汰。另请参见对--column-names 选项的介绍。

- --skip-line-number 或 -L

  -L 形式已经被淘汰。另请参见对--line-numbers 选项的介绍。

- `--table` 或 `-t`（布尔）
  把输出报告显示为表格形式，即各行之间用表格线分隔且纵向对齐。这是运行在非批处理模式中的 `mysql` 程序的默认输出格式。
- `--tee=`*`file_name`*
  把全部输出信息的一份副本追加到指定文件的末尾。可以用`--disable-tee` 选项来禁用这项功能。这个选项不能在批处理模式下工作。
- `--unbuffered` 或 `-n`（布尔）
  每执行完一个查询命令，就对用来与服务器通信的缓冲区进行“冲洗”。
- `--vertical` 或 `-E`
  按纵向方式显示查询结果，即查询结果中的每行将被显示为一组输出行，每列为一个输出行，由一个列名和值组成。每组输出行的第一行显示那条记录在结果集里的序号。如果查询命令检索出来的记录很长，把它们显示为纵向格式将增加可读性。
  如果没有给出这个选项，但后来又想使用纵向显示格式，就需要使用“`\G`”而不是“`;`”或“`\g`”来作为该查询命令的结束符。
- `--wait` 或 `-w`
  如果无法与服务器建立连接，则等待并重试。
- `--xml` 或 `-X`（布尔）
  生成 XML 输出。

F.5.3　与**mysql**有关的变量

下面这些与 `mysql` 程序有关的变量都可以按 F.2.1 节中第 2 小节介绍的步骤设置。

- `connect_timeout`
  如果在经过这么多秒之后还没有建立起与服务器的连接，则放弃这次尝试。这个变量的默认值为 0。
- `max_allowed_packet`
  服务器与客户通信时使用的缓冲区的最大长度。默认值是 16MB，最大值是 1GB。
- `max_join_size`
  如果 `mysql` 程序在启动时使用了`--safe-updates` 选项，这个变量的值就是联结执行的数据行的最大个数。如果服务器认为需要检查的行数大于 `max_join_size`，它就会拒绝联结。默认值是 1 000 000。
- `net_buffer_length`
  服务器与客户通信时使用的缓冲区的初始长度。这个缓冲区可以扩张到 `max_allowed_packet` 个字节长。默认值是 16 KB。
- `select_limit`
  如果程序在启动时使用了`--safe-updates` 选项，这个变量的值就是 `SELECT` 语句返回的数据行的最大个数。默认值是 1000。

F.5.4　**mysql**命令

除允许向服务器发送 SQL 语句外，`mysql` 程序本身还有一些内部命令。在给出这些命令的时候，

你必须全都写在同一行上。这些命令中的大多数都有一个长格式形式（一个单词）和一个短格式形式（一个反斜线加一个字符）。长格式命令不区分字母的大小写情况，短格式命令则必须按下面有关内容中的字母大小写形式写出。你可以在长格式命令的末尾加上一个分号作为结束符，但不是必须这样做。短格式命令中不应使用分号。

如果禁用了“允许 mysql 程序的内部命令的长格式形式出现在任何输入行的开头部分”功能（例如，通过--disable-named-commands 选项），长格式命令就只允许出现在主提示符 mysql>处。

- clear 或 \c
  清除（撤销）当前查询。所谓“当前查询”指的是你正在敲入的那条查询语句。那些已经被发送给服务器或者 mysql 程序已经开始显示其查询结果的查询是无法用这个命令来撤销的。
- connect [*db_name* [*host_name*]] 或 \r [*db_name* [*host_name*]]
  连接（或者再次连接）指定主机上的指定数据库。如果没有给出数据库名或主机名，则使用当前 mysql 会话中最近一次用过的值。
- delimiter *str* 或\d *str*
  设置语句分隔符，默认的分隔符是分号。存储程序分析器只能识别分号分隔符，所以在定义存储程序时这个语句可以用来为 mysql 重新定义分隔符，如 4.1 节中所述。
  最好避免在分隔符中使用反斜线，因为 MySQL 会把反斜线当做转义字符。
- edit 或 \e
  编辑当前查询。mysql 程序将依次检查环境变量 EDITOR 和 VISUAL 的值以决定要使用哪一个编辑器。如果这两个环境变量都没有定义，mysql 程序将使用 vi 编辑器。这个选项在 Windows 系统上不可用。
- ego 或 \G
  把当前查询发送到服务器并按纵向格式显示查询结果。
- exit
  与 quit 命令作用相同。
- go 或 \g
  把当前查询发送到服务器并显示其查询结果。
- help 或 \h 或 ?或\?
  显示帮助信息，描述可用的 mysql 命令。
  如果 mysql 数据库中的帮助表已经加载，也可以使用 help 来寻求服务器端帮助：使用 help contents 获取一列帮助类别、使用 help category 获取特定类型的帮助、使用 help keyword 获取有关特定关键字的帮助（如 SELECT 或 UPDATE）。有关加载帮助表的说明请参见 A.4.4 节的第 4 小节。
- nopager 或 \n
  禁用分页显示机制，把输出发送到标准输出设备。这个命令在 Windows 系统上不可用。
- notee 或 \t
  不把输出内容追加到 tee 文件的末尾。
- nowarning 或\w
  不要自动显示每个语句生成的任何警告。这个命令是从 MySQL 5.0.6 引入的。
- pager [*program*] 或 \P [*program*]
  把输出内容发送到 program 参数或者 PAGER 环境变量（如果设置了这个变量而且没有给出

program）所指定的分页显示程序。这个命令在 Windows 上不可用。

- print 或 \p
  显示当前查询的文本（只显示查询命令本身，不显示执行这条查询命令而获得的结果）。
- prompt *arguments* 或 \R *arguments*
  重新定义主提示符 mysql>。新提示符字符串将由从 prompt 关键字后面第一个空格开始的所有字符（包括其他空格）构成。这个字符串可以包含一些特殊的序列，具体情况见 F.5.5 节。如果想把提示符恢复为默认值，请发出一条不带任何参数的 prompt 或\R 命令。
- quit 或 \q
  退出 mysql 程序。
- rehash 或 \#
  重新对数据库、数据表、数据列的名字进行散列计算，计算结果将用于实现这些名字的自动补足功能。另请参见对--auto-rehash 选项的介绍。
- source *file_name* 或 \. *file_name*
  从指定文件里读取并执行语句。注意：Windows 路径名中的反斜线字符（\）必须双写或者写成斜线字符（/）。
- status 或 \s
  检索并显示来自服务器的状态信息，比如服务器的版本、默认数据库、当前连接是否安全，等等。
- system *command* 或 \! *command*
  用默认 shell 解释器来执行 command 命令。这个命令在 Windows 系统上不可用。
- tee [*file_name*] 或 \T [*file_name*]
  把输出内容追加到指定文件的末尾。
- use *db_name* 或 \u *db_name*
  把指定数据库选定为当前的默认数据库。
- warnings 或\W
  自动显示每个语句生成的任何警告。这个命令是从 MySQL 5.0.6 引入的。

F.5.5　mysql程序的提示符定义序列

在默认情况下，mysql 程序的主提示符是 mysql>，但我们可以通过环境变量 MYSQL_PS1、--prompt 选项或者 prompt 命令来重新定义。比如说，如下所示的 prompt 命令将使默认数据库的名字出现在提示符里，当我们改用另一个默认数据库时，提示符将发生相应的变化：

```
% mysql
mysql> prompt \d>\_
PROMPT set to '\d>\_'
(none)> USE sampdb;
Database changed
sampdb> USE test;
Database changed
test>
```

跟在 prompt 关键字后面的是提示符定义字符串。在提示符定义里，以反斜线（\）开头的转义序列代表特殊的提示符选项。比如说，“\d”和“_”序列分别代表默认数据库的名字和一个空格。（如

果使用环境变量 MYSQL_PS1 或--prompt 选项来设置提示符，在给出提示符字符串的时候，可能需要连续敲入两个反斜线字符。）表 F-6 提供了一份可用选项的完整清单。

表 F-6

转义序列	含　义
\c	当前输入行数
\d	默认数据库的名字。如果尚未选定数据库，则是（none）
\D	完整的日期和时间
\h	当前主机
\l	当前分隔符（从MySQL 5.0.25/5.1.12起）
\m	分钟
\o	月份数字
\O	月份名称，3个字母
\p	当前端口号、套接字文件名、命名管道的名字或共享内存的名字
\P	时间值的am或pm标志
\r	小时（12小时制）
\R	小时（24小时制）
\s	秒
\S	分号
\t	制表符
\u	当前用户名，不带主机名
\U	当前用户名，带主机名
\v	服务器的版本号
\w	星期几，3个字母
\y	年（2位数字）
\Y	年（4位数字）
\'	单引号
\"	双引号
_	空格字符
\	空格字符（这个转义序列是一个反斜线加一个空格）
\\	反斜线字符
\n	换行符
\x	字符x，x是没有在上面列出的任何字符

F.6 mysql.server

mysql.server 通过调用 mysql_safe 脚本来启动或是关停 mysqld 服务器。它是一个 shell 脚本，适用于 Unix 系统。

mysql.server 脚本支持 start 和 stop 两个命令行参数：

```
mysql.server start
mysql.server stop
```

mysql.server 脚本通常被安装在 Unix 系统的某个运行级目录（如/etc 的某个下级子目录）里。（安装在系统里的 mysql.server 脚本通常会被命名为“mysql”而不是“mysql.server”。）这种安

排的好处是：在开机时，系统将自动以 start 参数来调用这个脚本去启动服务器；在系统关机时，系统又将自动地以 stop 参数来调用这个脚本去关闭服务器。当然，也可以手动启动或者关停服务器，只要在命令行上给出正确的参数就行了。

mysql.server脚本支持的选项

mysql.server 脚本支持的标准 MySQL 选项很少。它不从命令行读取任何标准的选项。在选项文件中的[mysql.server]选项组里，它读取 basedir、datadir 和 pid_file 选项并把它们传递给 mysql_safe 脚本。从 MySQL 5.0.40/5.1.17 版本开始，新增了一个 service_startup_timeout=n 选项，设置这个脚本等待服务器启动的时间是多少秒，默认值是 900，0 值的含义是“不等待”，负值的含义是“永远等待”。

F.7　mysql_config

mysql_config 是以 C 语言开发 MySQL 应用程序时使用的一个工具程序，能让你获得在编译 C 语言源文件或链接 MySQL 开发库时必须用到的标志。

```
mysql_config [ options ]
```

mysql_config独有的选项

- --cflags
  显示访问 MySQL 头文件所必需的包含目录标志以及其他可能用到的 C 编译标志。
- --embedded 或 --embedded-libs
  这两个选项是--libmysqld-libs 选项的同义词。
- --include
  显示用来访问 MySQL 头文件的包含目录标志。
- --libmysqld-libs
  显示链接嵌入式服务器开发库 libmysqld 所必需的库标志。
- --libs
  显示链接客户端库所必需的库标志。
- --libs_r
  显示用来链接线程安全客户端库的库标志。
- --plugindir
  显示默认的插件子目录。这个选项是从 MySQL 5.1.24/6.0.5 版开始引入的。
- --port
  显示默认的 MySQL TCP/IP 端口号。
- --socket
  显示默认的 Unix 套接字文件的路径名。
- --version
  显示 MySQL 版本字符串。

F.8　mysql_install_db

mysql_install_db 脚本能够创建服务器的数据目录，对包含各种权限数据表的 mysql 数据库进

行初始化，创建空白的 test 数据库。

```
mysql_install_db [options]
```

mysql_install_db 脚本将在权限数据表里创建几个基本的用户账户，比如 root 账户和匿名账户等。本书第 12 章对这些基本账户以及如何利用口令来加强数据库系统安防水平做了一些探讨。

mysql_install_db Windows 系统上不可用，也不是必需的，因为 Windows 发行版本包含预初始化的数据目录。

F.8.1 mysql_install_db脚本支持的标准选项

--help --user

--help 选项是从 MySQL 5.0.48/5.1.21 版开始增加的。

在 Unix 系统上，--user 选项的作用是让服务器使用给定用户的登录账户启动运行。这么做的好处是：当你以 Unix 系统的 root 用户身份运行 mysql_install_db 脚本时，可以确保服务器创建的目录和文件都以给定用户为属主。

F.8.2 mysql_install_db独有的选项

你可以在命令行上使用本节提到的选项。只要在选项文件中的[mysqld]组里使用一些相应的设置项，就能对这些值进行设定。从 MySQL 3.23.29 版本开始，这个脚本还会读取[mysql_install_db]选项组，这就使 mysql_install_db 脚本所独有而 mysqld 程序却不知道的选项（比如--ldata 和--force）用起来更方便了。

Mysql_install_db 将任何未识别的选项传递给 mysqld。

- --basedir=*dir_name*

  把这个子目录用作 MySQL 基本目录。
- --datadir=*dir_name* 或 --ldata=*dir_name*

  把这个子目录用作 MySQL 数据目录。
- --force

  强行运行，不管当前主机名能否得到确定。此时，权限数据表里的记录项将使用主机的 IP 数字地址来创建，这意味着你只能使用 IP 数字而不是主机名（本地主机 localhost 是个例外）去使用客户程序。
- --skip-name-resolve

  在权限数据表里只使用数字形式的 IP 地址，不使用主机名。万一你的 DNS 服务器不够好或根本就没有，这个选项就非常有用了。

F.9 mysqladmin

mysqladmin 程序通过与服务器通信可以完成各种管理工作。你可以利用 mysqladmin 程序去获取服务器操作信息、控制服务器操作、设置口令、创建或丢弃数据库。

F.9.1 mysqladmin支持的标准选项

```
--character-sets-dir    --host        --silent
--compress              --password    --socket
--debug                 --pipe        --user
--debug-check           --port        --verbose
--debug-info            --protocol    --version
```

```
--default-character-set    --set-variable
--help                     --shared-memory-base-name
```

--debug-info 和--debug-check 分别是从 MySQL 5.1.14 和 5.1.21 引入的。

如果使用了--silent 选项，mysqladmin 程序在无法与服务器连接时就会默默地退出执行。--verbose 选项使 mysqladmin 程序在执行某些命令时显示更多的信息，它还支持标准的 SSL 选项。

F.9.2 **mysqladmin**独有的选项

- --count=*n* 或 -c *n*

  与--sleep 选项一起使用，给出 mysqladmin 程序将要循环执行的次数。如果给出了--sleep 但没给出--count，将永远循环下去，除非你中断它。
- --force 或 -f（布尔）

  这个选项有两个效果：其一，让 mysqladmin 程序在执行 drop db_name 命令时不进行确认；其二，当你在命令行上同时给出多个命令时，mysqladmin 程序将执行每个命名不管是否有执行出错的。在正常情况下，mysqladmin 程序将在第一次出错后退出执行。
- --no-beep 或-b（布尔）

  让系统在发生错误时不发出提示音。这个选项是从 MySQL 5.1.17 版开始引入的。
- --relative 或 -r（布尔）

  与--sleep 选项一起使用，显示 mysqladmin 程序前后两次执行结果的不同之处。这个选项目前只能与 extend-status 命令一起使用才有效果。
- --sleep=*n* 或 -i *n*

  每隔 *n* 秒重复执行一次你在命令行上给出的命令。
- --vertical 或 -E（布尔）

  这个选项与--relative 选项的功能相同，但将把输出内容按纵向格式显示。
- --wait[=*n*] 或 -w[*n*]

  如果无法与服务器连接，则等待 *n* 秒之后再重试。如果没有设定，*n* 的默认值将是 1。如果使用-w 来设置 *n* 值，它们之间不允许有空格，否则 *n* 值将无法得到正确的解释。

F.9.3 与**mysqladmin**有关的变量

下面这些与 mysqladmin 程序有关的变量都可以按 F.2.1 节中第 2 小节的步骤进行设置。

- connect_timeout

  如果在经过了 connect_timeout 秒之后还没连接上服务器，则放弃本次尝试。默认值为 43 200。
- shutdown_timeout

  要求 shutdown 命令必须在 shutdown_timeout 秒内成功地关闭 MySQL 服务器。默认值为 36 000。

F.9.4 **mysqladmin**命令

可以在命令行上的选项后面给出一个或者多个下列命令。在不引起二义的前提下，允许只写出命令的前缀。比如说，processlist 命令允许简写为 process 或 proc，但不允许简写为 p。

部分命令有着与之功能相同的 SQL 语句，具体情况见有关条目说明。对那些 SQL 语句的详细说明请参见附录 E。

- create *db_name*

  以给定名字创建一个新数据库。这个命令与 CREATE DATABASE db_name 语句等价。

- debug
  让服务器将调试信息转储到错误日志。
- drop *db_name*
  删除指定的数据库以及该数据库里的数据表。如果没有使用--force 选项，mysqladmin 程序将要求确认这个命令。这个命令与 DROP DATABASE db_name 语句等价。
- extended_status
  显示服务器状态变量的名字和值。这个命令与 SHOW STATUS 语句等价。
- flush_hosts
  清空主机缓存。这个命令与 FLUSH HOSTS 语句等价。
- flush_logs
  清空（关闭再打开）日志文件。这个命令与 FLUSH LOGS 语句等价。
- flush_privilegs
  重新加载权限数据表。这个命令与 FLUSH PRIVILEGES 语句等价。
- flush_status
  对状态变量进行清零（把多个计数器重置为 0）。这个命令与 FLUSH STATUS 语句等价。
- flush_tables
  清空数据表缓存。这个命令与 FLUSH TABLES 语句等价。
- flush_threads
  清空线程缓存。
- kill *id*,*id*,...
  终止执行指定的服务器线程。如果同时给出了多个线程 ID 号，它们之间将不允许有任何空格，以免它们被误认为是跟在 kill 命令后的其他命令。可以用 mysqladmin processlist 命令查出当前都有哪些线程正在运行。这个命令与使用 KILL 语句逐个终止各个线程的做法等价。
- old-password *new_password*
  这个命令和 password 命令相似，但这个命令会把口令保存为 MySQL 4.1 系列版本里使用的老格式。
- password *new_password*
  修改服务器在你请求连接时为你核定的账户的口令。（你能够使用这个账户连接服务器的事实已足以证明你知道现有的口令。）新口令将被设置为 *new-password*。这个命令和 SET PASSWORD 语句相似。
  在 Unix 上，你可以在 mysqladmin 命令中使用单引号或双引号来引述口令，如果口令包含的字符会被命令解释器认为是特殊字符的话；在 Windows 上，只能使用双引号。Windows 命令解释器不会把单引号识别为参数引述字符。如果使用了单引号，它们将成为口令的一部分。
- ping
  检查 MySQL 服务器是否正在运行。
- processlist
  列出当前正在执行的服务器进程。这个命令与 SHOW PROCESSLIST 语句等价。如果还使用了--verbose 选项，那么这个命令将与 SHOW FULL PROCESSLIST 语句等价。
- refresh
  这个命令将清空数据表缓存和各个权限数据表，同时还会先关闭再打开日志文件。如果服务器

是复制机制中的主服务器，这个命令将使它删除在二进制日志索引文件里列出的二进制日志文件并把其索引截短为 0。如果服务器是从服务器，这个命令将使它忘记自己在主日志里的位置。

- reload
  重新加载权限数据表。这个命令与 FLUSH PRIVILEGES 语句等价。
- shutdown
  关闭 MySQL 服务器。
- start_slave
  启动一个复制从服务器。这个命令与 START SLAVE 语句等价。
- status
  按简短格式显示服务器的状态信息。
- stop_slave
  关闭一个复制从服务器。这个命令与 STOP SALVE 语句等价。
- variables
  显示服务器各变量的名字和值。这个命令与 SHOW GLOBAL VARIABLES 语句等价。（没有与 SHOW SESSION VARIABLES 语句等价的 mysqladmin 命令，因为这毫无意义。）
- version
  检索并显示服务器的版本信息字符串，这个字符串与 VERSION()函数的返回值完全相同（请参见附录 C）。

F.10 mysqlbinlog

mysqlbinlog 程序将以可读格式显示二进制日志文件的内容：

```
mysqlbinlog [ options ] file_name ...
```

mysqlbinlog 程序的默认行为是不连接服务器而直接读取本地日志文件。它也可以连接到一个服务器并请求它把日志文件通过连接发送过来，详见关于--read-from-remote=server 选项的描述。

二进制日志的格式一直处于变化中。为避免兼容性问题，你使用的 mysqlbinlog 程序的版本至少应该不低于服务器的版本。

mysqlbinlog 程序还可以读取从服务器创建的中继日志文件，因为中继日志和二进制日志的格式是完全一样的。

F.10.1 mysqlbinlog程序支持的标准选项

```
--character-sets-dir      --help              --protocol
--debug                   --host              --socket
--debug-check             --password          --user
--debug-info              --port              --version
```

--character-set-dir 选项是从 MySQL 5.0.3 版开始增加的，--debug-check 和--debug-info 选项是从 MySQL 5.1.21 版开始新增加的。

F.10.2 mysqlbinlog程序独有的选项

- --base64-output [=*value*]
  是否（以及何时）使用 base-64 编码格式的 BINLOG 语句作为输出。这个选项是从 MySQL 5.0.5 版开始增加的，当时它只是一个布尔选项。从 5.1.24 版开始，它的可取值是 auto（只在确有

必要时才使用 base-64 编码)、always（总是使用）或 never（不使用）。如果没有给出这个选项，默认值是 auto；如果给出了这个选项但没有赋值，默认值是 always。

- --database=*db_name* 或 -d *db_name*
  从日志文件里只提取与指定数据库有关的语句。这个选项只在读取本地日志时有效。
- --disable-log-bin 或 -D（布尔）
  在输出里增加语句来禁用针对数据更新语句的二进制日志功能，这是为了防止在恢复数据库或数据表时把数据更新语句再次记载到二进制日志里。
- --force-if-open 或 -F（布尔）
  强行读取二进制日志文件，哪怕它们没有正确关闭（或正在被使用）。这个选项是从 MySQL 5.1.15 版开始新增加的。
- --force-read 或 -f（布尔）
  这个选项控制着 mysqlbinlog 程序在二进制日志里读到一个它无法识别的事件时会采取什么行动。在默认的情况下，它将退出运行。如果启用了这个选项，mysqlbinlog 程序将在记录一条警告消息并丢弃那个事件后继续执行。
- --hexdump 或 -H（布尔）
  在输出里包括一份十六进制/ASCII 格式的事件备份。这个选项是从 MySQL 5.0.16 版开始新增加的。
- --local-load=*dir_name* 或-l *dir_name*
  为处理 LOAD DATA LOCAL 语句而创建的临时文件将保存在这个子目录里。
- --offset=*n* 或 -o *n*
  跳过日志文件中的前 *n* 项记录。
- --position=*n* 或 -j *n*
  这个选项已被--start-position 选项取代。
- --read-from-remote-server 或 -R（布尔）
  通过与服务器建立一条网络连接并请求通过此连接发送日志来读取二进制日志文件。为此，在给出--read-from-remote-server 的同时，还需要根据具体情况给出--host、--password、--protocol、--socket 和--user 选项以设定连接参数。如果没有给出--read-from-remote-server 选项，上述选项都将被忽略。
- --result-file=*file_name* 或 -r *file_name*
  把输出内容写到指定的文件里去。
- --server-id=*n*
  只输出与 ID 值等于 *n* 的服务器有关的事件。这个选项是从 MySQL 5.1.4 版开始新增加的。
- --set-charset=*charset*
  在输出里增加一条 SET NAMES 语句。这个选项是从 MySQL 5.0.23/5.1.12 版开始新增加的。
- --short-form 或 -s
  只显示日志里记载的语句，不显示日志里记载的与那些语句有关的其他信息，也不显示基于数据行的事件。
- --start-datetime=*date_time*
  从发生时间等于或晚于 *date_time* 的那些事件开始读取二进制日志事件。*date_time* 值必须以一种合法的 DATETIME 格式给出，代表用来运行 mysqlbinlog 程序的那台主机所在的地理

时区中的时间。如有必要，请根据命令解释器对 *date_time* 值进行转义处理。

- --start-position=*n*
  从命令行上列出的第一个日志文件里的给定位置开始读取二进制日志事件。
- -- stop-datetime=*date_time*
  当事件发生时间等于或晚于 *date_time* 时，停止读取二进制日志事件。*date_time* 值必须以一种合法的 DATETIME 格式给出，代表用来运行 mysqlbinlog 程序的那台主机所在的地理时区中的时间。如有必要，请根据命令解释器对 *date_time* 值进行转义处理。
- --stop-position=*n*
  当到达命令行上列出的最后一个日志文件的给定位置时，停止读取二进制日志事件。
- --to-last-log 或 -R（布尔）
  在从服务器读取日志文件时（这需要用到--read-from-remote-server 选项），这个选项将使程序一直读到该服务器上的最后一个二进日志文件，而不是在命令行上列出的最后一个日志文件的末尾。--to-last-log 选项可以确保那台服务器上的所有二进制日志信息全被读取出来。（不过，如果你把读取出来的事件发送回同一个服务器的话，使用这个选项将导致一个无限循环。）

F.10.3　与**mysqlbinlog**程序有关的变量

下面是与 mysqlbinlog 程序有关的变量，它们可以按照 F.2.1 节的第 2 小节里给出的步骤进行设置。

- open_files_limit
  保留的文件描述符的个数，默认值是 64。

F.11　**mysqlcheck**

mysqlcheck 是用来检查和修复数据表的客户程序。它为 CHECK TABLE、ANALYZE TABLE、OPTIMIZE TABLE、REPAIR TABLE 等语句提供了一个命令行接口。它与 myisamchk 程序有几分相似，但它是在服务器运行时使用的，对非 MyISAM 表有一些支持。mysqlcheck 程序发送管理查询命令到服务器去执行，这与 myisamchk 程序形成了鲜明的对照，myisamchk 直接在数据表文件上操作，所以必须由你协调服务器对关数据表的访问或者干脆停止服务器的运行。

mysqlcheck 程序的各个选项全都可以用在 MyISAM 数据表上。mysqlcheck 程序还可以对 InnoDB 数据表进行检查分析。

mysqlcheck 程序有 3 种运行模式：

```
mysqlcheck [options] db_name [tbl_name] ...
mysqlcheck [options] --databases db_name ...
mysqlcheck [options] --all-databases
```

在第一种模式里，mysqlcheck 程序将检查指定数据库里的指定数据表。如果没有给出数据表的名字，mysqlcheck 将检查数据库里的所有数据表。在第二种模式里，mysqlcheck 程序将把所有的参数都解释为数据库的名字，依次检查各数据库里的所有数据表。在第三种模式里，mysqlcheck 程序将依次检查所有数据库里的所有数据表。

F.11.1　**mysqlcheck**支持的标准选项

```
--character-sets-dir     --help          --shared-memory-base-name
--compress               --host          --silent
--debug                  --password      --socket
```

```
--debug-check              --pipe                     --user
--debug-info               --port                     --verbose
--default-character-set    --protocol                 --versiont
```

--debug-check 和--debug-info 是从 MySQL 5.1.21 开始引入的。

mysqlcheck 程序还支持各种标准的 SSL 选项。

F.11.2 mysqlcheck独有的选项

下列选项控制着 mysqlcheck 程序将如何对数据表进行处理。在介绍完这选项之后，我们还将介绍这些选项与 SQL 语句的等价对应关系。

- --all-databases 或 -A（布尔）
  检查所有数据库里的所有数据表。
- --analyze 或 -a
  发出 ANALYZE TABLE 语句，分析数据表。（比如说，这个选项将分析键值的分布情况。）分析结果将有助于更快地完成基于索引的查找和联结操作。
- --all-in-1 或 -1（布尔）
  如果没有使用这个选项，mysqlcheck 程序将为每个表分别发出一个查询。如果使用了这个选项，mysqlcheck 程序将按数据库对数据表归组，用同一个语句对同一个数据库里的所有数据表进行检查。
- --auto-repair（布尔）
  如果检查发现某些数据表有问题，程序将在这次检查完毕后再执行一遍以自动修复它们。
- --check 或 -c
  发出 CHECK TABLE 语句，检查数据表中有无错误。如果没有明确告诉 mysqlcheck 说你想让它做些什么，这将是它的默认行为。
- --check-only-changed 或 -C
  只检查自从上次检查后发生改变的表，或者未正确关闭的表。
- --check-upgrade 或-g
  检查数据表是否与 MySQL 当前版本兼容，更新后是否有用。采用--auto-repair 的话，如果发现不兼容，将自动修复。这是从 MySQL 5.0.19/5.1.7 开始引入的。
- --databases 或 -B （布尔）
  把所有的参数都解释为数据库的名字，检查各数据库里的所有数据表。
- --extended 或 -e（布尔）
  对数据表进行全面检查。如果与--repair 选项联合使用，mysqlcheck 将使用一种比单独给出--repair 选项时更全面但也更慢的修复方法。
- --fast 或 -F（布尔）
  只检查没有被正确关闭的数据表。
- --fix-db-names（布尔）
  检查数据库的名字并根据 MySQL 5.0 和 5.1 系列版本的不同要求对它们进行编码转换。这个选项是从 MySQL 5.1.7 版开始引入的。
- --fix-table-names（布尔）
  对数据表的名字进行检查并根据 MySQL 5.0 和 5.1 系列版本的不同要求对它们进行编码转换。这个选项是从 MySQL 5.1.7 版开始引入的。

- --force 或 -f（布尔）
  强行继续执行，不管是否出错。
- --medium-check 或 -m
  对数据表进行中级检查。此时使用的数据表检查方法要比给出--extended 选项时的快一些，但不那么全面。这个检查模式已经足以应付大多数场合。
- --optimize 或 -o
  发出 OPTIMIZE TABLE 语句，对数据表进行优化。
- --quick 或 -q （布尔）
  对于数据表检查操作，这个选项将省略对数据行中的链接进行检查的步骤。如果是与--repair 选项联合使用，这个选项将只修复索引文件而不触及数据文件。重复写两遍这个选项与只写一遍的效果是一样的，这是与 myisamchk 程序的又一个不同之处（对于后者，把这个选项写两遍的效果与写一遍的效果是不一样的）。
- --repair 或 -r
  发出 REPAIR TABLE 语句，对数据表进行修复。这个修复模式能够纠正绝大多数问题，但对唯一化索引中的重复键值问题无能为力。
- --tables
  覆盖--databases 选项，使随后的任何参数都被解释为表名。
- --use-frm（布尔）
  与--repair 选项一起使用，让数据表修复操作使用.frm 文件重新初始化索引文件，确定如何解释数据文件和重建索引文件。这个选项主要用在索引文件已丢失或者被损坏的场合。但它只应被当做最后的重排序，并且只在当前 MySQL 版本与创建表时所用版本一样时才能使用。
- --write-binlog（布尔）
  把 ANALYZE TABLE、OPTIMIZE TABLE 和 REPAIR TABLE 语句记载到二进制日志里（这意味着它们也将被发送到从服务器去）。这个选项是默认启用的，你可以用--skip-write-binlog 选项禁用。这个选项是从 MySQL 5.1.18 版开始引入的。

mysqlcheck 程序的选项与相应的 SQL 命令的等价对应关系见表 F-7 至表 F-10。

数据表检查选项（只适用于 MyISAM 和 InnoDB 数据表），见表 F-7。

表　F-7

选　　项	相应的语句
--check	CHECK TABLE *tbl_list*
--check-only-changed	CHECK TABLE *tbl_list* CHANGED
--extended	CHECK TABLE *tbl_list* EXTENDED
--fast	CHECK TABLE *tbl_list* FAST
--medium-check	CHECK TABLE *tbl_list* MEDIUM
--quick	CHECK TABLE *tbl_list* QUICK

对于 InnoDB 表，表 F-7 中的所有选项都被当做--check，InnoDB 不支持不同类型的检查。

数据表分析选项（只适用于 MyISAM 和 BDB 数据表），见表 F-8。

表 F-8

选 项	相应的语句
--analyze	ANALYZE TABLE *tbl_list*

数据表修复选项（只适用于 MyISAM 数据表），见表 F-9。

表 F-9

选 项	相应的语句
--repair	REPAIR TABLE *tbl_list*
--repair --quick	REPAIR TABLE *tbl_list* QUICK
--repair --extended	REPAIR TABLE *tbl_list* EXTENDED
--repair --use-frm	REPAIR TABLE *tbl_list* USE_FRM

数据表优化选项（只适用于 MyISAM 数据表），见表 F-10。

表 F-10

选 项	相应的语句
--optimize	OPTIMIZE TABLE *tbl_list*

F.12 mysqld

mysqld 就是 MySQL 服务器程序。它是客户程序访问数据库的桥梁，要是服务器没有运行，客户程序就无法使用该服务器所管理的数据库。在启动的时候，mysqld 将打开一些要监听的网络接口并开始在等待客户连接。mysqld 是一个多线程的程序，它将使用不同的线程来处理各客户连接，使多个客户程序可以并发处理。对数据库进行写操作的查询都将以原子化方式处理，也就是说，当服务器开始执行一个这样的查询时，任何涉及它正在处理的数据的查询都将被阻塞，直到当前查询执行完毕为止。比如说，两个客户程序不可能同时修改同一数据表里的同一个数据行。

mysqld 程序最常见的启动方式很简单，你只需写出服务器程序的名字和想使用的选项就可以了，如下所示：

```
mysqld [options]
```

在基于 Windows 的系统上，还可以把服务器安装并运行为一项服务。下面第一条命令让服务器在系统开机启动时自动运行，第二条命令则删除了这项服务：

```
C:\> C:\mysql\bin\mysqld --install
C:\> mysqld --remove
```

安装命名使用了服务器的完整路径名。如果服务器安装在了不同的位置，将相应地修改路径名。默认的服务名是 MySql。还可以在选项的后面另行指定一个服务名，如下所示：

```
C:\> C:\mysql\bin\mysqld --install service_name
C:\> mysqld --remove service_name
```

这就使多个 MySQL 服务器能够以不同的服务名同时运行。如果没有给出 service_name 参数或者服务名不为 MySQL，MySQL 服务器就将以 MySQL 作为服务名，并会从标准选项文件读取选项文件组 [mysqld]。如果你给出了 service_name 参数，MySQL 服务器就将以该参数作为自己的服务名，并从标准选项文件读取选项文件组 [service_name] 和 [mysqld]。

还可以在服务名的后面用--defaults-file 选项再额外指定一个选项文件，这样，服务器将在启

动时读取该文件里的选项。如下所示：

```
C:\> C:\mysql\bin\mysqld --install service_name --defaults-file=file_name
```

在上面这种场合，service_name 参数不允许省略。

刚才对--install 选项的讨论也适用于--install-manual 选项。

F.12.1　mysqld支持的标准选项

```
--character-sets-dir      --port                    --user
--debug                   --shared-memory-base-name --verbose
--help                    --socket
```

--help 选项只显示一个简短的用法消息，若查看完整的帮助信息，使用下面的命令：

```
% mysqld--verbose--help
```

mysqld 程序还支持各种标准的 SSL 选项。

注意，虽然 MySQL 现在支持使用--socket 选项，但目前仍不支持你使用相应的短格式（-S）。在 Windows 上，如果服务器支持命名管道连接，--Socket 将设置管道名称。

在 Unix 系统上，如果使用了--user 选项，服务器将使用该账户的用户名或 ID 数字来运行。此时，在启动的时候，服务器将从口令文件里查出该账户的用户 ID 和用户组 ID，并把它们用作它自己的用户 ID 和用户组 ID。如此启动的服务器在运行时将不再具备 root 权限，它的权限将由指定账户来决定。（不过，要想让--user 选项起作用，服务器就必需按 root 用户来启动，因为只有这样它才能改变自己的用户 ID，否则会报告警告信息。）

F.12.2　mysqld独有的选项

下列选项是服务器的通用选项。在随后的几个小节里，我们将依次介绍特定于 Windows 系统、InnoDB 和复制机制的选项。

- --allow-suspicious-udfs（布尔）

  允许 MySQL 服务器加载老式用户定义函数（User-Defined Function，UDF），老式用户定义函数可能只定义了与函数名有关的符号而没有对与标准化底层支持例程有关的符号做出任何定义。为避免把普通函数错当成老式用户定义函数来加载，这个功能默认禁用。这个选项是从 MySQL 5.0.3 版开始引入的。
- --ansi 或 -a

  使服务器在遇到某些语法时按标准 SQL 行为而不是按 MySQL 标准来采取行动。这个选项将使 MySQL 服务器的行为与标准兼容。

  这个选项相当于你在使用--sql-mode 选项的同时还给出了 REAL_AS_FLOAT、PIPES_AS_CONCAT、ANSI_QUOTES、IGNORE_SPACE 和 ONLY_FULL_GROOUP_BY 模型值。
- --basedir=*dir_name* 或 -b *dir_name*

  MySQL 安装目录的路径名。以相对路径给出的很多其他路径是以这个子目录作为顶级目录的。
- --big-table

  使 MySQL 服务器能够对大结果集进行处理，它将把临时结果全都保存到磁盘上而不是放在内存里。如果没有使用这个选项，那么当没有足够的内存来容纳大结果集时，就经常会发生“table full”（数据表满）错误。这个选项已经不再是必需的了，因为服务器会自动保存结果到磁盘。

- --bind-address=*ip_addr*
  绑定到给定的 IP 地址。在一般情况下，mysqld 在它将运行的那台主机上有一个默认的绑定地址。如果主机有多个地址，就需要用这个选项来另行绑定一个地址了。
- --bootstrap
  这个选项是你第一次安装 MySQL 时使用的。
- --character-set-client-handshake（布尔）
  告诉服务器使用客户提供的字符集信息。这个选项是默认启用的。--skip-character-set-client-handshake 选项将导致那些信息被忽略——MySQL 4.0 系列版本就是那么做的。这个选项始见于 MySQL 5.0.13 版。
- --character-set--filesystem = *charset*
  设置 character-set--filesystem 系统变量。这个选项是从 MySQL 5.0.19/5.1.6 版开始引入的。
- --character-set-server = *charset* 或-C *charset*
  服务器级的默认字符集。
- --chroot = *dir_name* 或-r *dir_name*
  让 MySQL 服务器在运行时把指定的子目录作为它的根目录。请参阅 chroot()函数的 Unix 帮助手册页以了解更多有关 chroot()环境的信息。
- --collation-server = *collation*
  服务器级的默认排序方式。
- --concurrent-insert （布尔）
  允许 MyISAM 数据表上的并发插入操作，如果 MyISAM 数据表里没有空白块，并发插入操作将在检索现有数据行时把新记录添加到数据表的末尾。这个选项是默认启用的，若要禁用，使用--skip-concurrent-insert。
- --core-file
  当发生致命错误时，MySQL 服务器将在退出前生成一个内核文件（core file）。
- --datadir=dir_*name* 或 -h *dir_name*
  MySQL 数据目录的路径名。
- --default-character-set = *charcet*
  这个选项已过时，请使用--character-set-server 选项。
- -- default-collation = *collation*
  这个选项已过时，请使用--collation-server 选项。
- --default-storage-engine = *engine_name*
  默认使用的数据表存储引擎。*engine_name* 值应该是服务器所能支持的存储引擎之一，如 MyISAM 或 InnoDB 等。（*engine_name* 值不区分字母的大小写。）如果没有给出这个选项，MySQL 服务器将使用 MyISAM。
- --default-table-type = *engine_name*
  这个选项已过时，请使用--default-storage-engine 选项。
- --default-time-zone = *tz_name*
  把 MySQL 服务器的默认时区设置为 *tz_name*。有关时区值的讨论见 12.10.1 节。这个选项设置的是 time_zone 系统变量而不是 system_time_zone。

- --delay-key-write=*val*
  设定服务器处理 MyISAM 数据表上的键值延迟写操作的模式。val 的可取值有 3 种：(1) ON——根据每个数据表的具体情况来采取行动（即根据每个数据表在创建时使用的 DELAY_KEY_WRITE 选项值来决定是否延迟其键值的写操作），这是本选项的默认设置情况；(2) OFF——对任何 MyISAM 数据表都不进行键值延迟写操作；(3) ALL——对所有的 MyISAM 数据表进行键值延迟写操作。OFF 和 ALL 对所有的 MyISAM 数据表都一视同仁，无论它们在创建时使用的是哪一种 DELAY_KEY_WRITE 选项值。
  因为--delay-key-write=ALL 选项将对所有的 MyISAM 数据表都进行键值延迟写操作，而不管它们当初是如何创建的，所以，为了在 MyISAM 数据表上获得更高的性能，人们经常使用这个选项来启动复制机制中的从服务器。
- --des-key-file=*file_name*
  供 DES_ENCRYPT()和 DES_DECRYPT()函数使用的 DES 密钥所在文件的名字，这个文件的格式可以在附录 C 对 DES_ENCRYPT()函数的介绍内容里查到。
- --enable-locking
  已被--external-locking 选项取代。
- --enable-pstack（布尔）
  服务器会在执行出错时把符号栈的内容打印出来。
- --exit-info[=*n*] 或 -T[*n*]
  让 MySQL 服务器在退出时对信息进行调试。如果用-T 选项来设定 *n* 值，它们之间将不允许有间隔性的空格，否则，服务器将无法对 *n* 值进行正确的解释。
- --external-locking（布尔）
  在某些系统（比如 Linux 系统）里，外部锁定机制（即文件系统级的锁定机制）是被默认禁用的。这个选项将启用这类系统中的外部锁定机制。
  外部锁定机制比较麻烦，因为它在某些系统上不工作，并且只对那些仅进行读操作的操作起作用，如数据表检查（参见 14.1 节）。
- --flush
  每完成一次修改操作，就把所有的数据表转储到磁盘上去。这将大大降低因系统崩溃而导致数据表损坏的可能性，但对系统的性能有着严重的影响。因此，应该只在不稳定的系统上才使用这个选项，它只适用于 MyISAM 数据表。
- --gdb
  设置信号处理器，为 gdb 调试工具的使用做准备。
- --general-log（布尔）
  启用常规日志功能，日志文件名由--log-output 选项指定，而--skip--general-log 选项将禁用这个日志。这个选项是从 MySQL 5.1.12 版开始引入的。
- --init-connect = *str*
  为每个客户在客户连接时执行的语句。*str* 值必须是一条或多条以分号分隔的 SQL 语句。这些语句只为不具备 SUPER 权限的客户执行。
- --init-file=*file_name*
  指定一个文件名，服务器将在启动时自动执行这个文件里的 SQL 语句。如果给出的 file_name 参数是一个相对路径，程序就将以数据目录为起点来对它作出解释。在这个文件里，每行只能

包含一条 SQL 语句。

- --isam（布尔）

  这个选项已经过时，因为 ISAM 存储引擎在 MySQL 5.0 系列版本里已不复存在。这个选项本身是在 5.1.14 版里删掉的。

- --language=*lang_name* 或 -L *lang_name*

  用指定语言向客户程序显示出错信息。*lang_name* 参数最常见的取值是 english（英语）和 german（德语），但也可以是某个用来存放语言文件的目录的绝对路径名。

- --large-pages（布尔）

  启用对大内存页面的支持功能。这个选项只在编译有这种支持的 MySQL 服务器里才会出现。这个选项是从 MySQL 5.0.3 版开始引入的。

- --lc-time-names = *locale_name*

  把 lc-time-names 系统变量设置为 *locale_name*。这个选项是从 MySQL 5.0.42/5.1.18 版开始引入的。

- --local-infile（布尔）

  启用或者禁用 LOAD DATA LOCAL 语句。调用--local-infile 选项将启用服务器端的 LOCAL 机制，调用--disable-local-infile 选项将禁用服务器端的 LOCAL。

- --log[=*file_name*] 或 -l[*file_name*]

  激活与常规日志有关的日志机制。常规日志记录着关于客户连接和 SQL 语句的一般性信息。日志文件名由--log-output 选定。如果你没有给定 *file_name* 参数，这个日志文件名就是数据目录中的 *HOSTNAME*.log，其中的 *HOSTNAME* 是服务器主机名。如果给出的 *file_name* 参数是一个相对路径，mysqld 将以数据目录为起点来对它作出解释。如果用-l 选项来设定 *file_name* 参数，它们之间将不允许有间隔性的空格；否则，*file_name* 值可能无法得到正确的解释。

- --log-bin[=*file_name*]

  启用二进制日志；*file_name* 指定二进制日志文件的基本名。如果你没有给定 file_name 参数，这类日志的文件名将是数据目录中的 *HOSTNAME*-bin.*nnnnnn*，其中 *HOSTNAME* 是该服务器主机名，*nnnnnn* 则是一个按 1 递增的序号（每创建一个新日志，就增加 1）。如果给出的 *file_name* 参数是一个相对路径，mysqld 将以数据目录为起点来对它作出解释。

- --log-bin-index=*file_name*

  启用二进制日志索引文件。如果没给出 *file_name*，默认值就是二进制日志文件的基本名，扩展名为.index。如果给出的 *file_name* 参数是一个相对路径，mysqld 将以数据目录为起点来对它作出解释。

- --log-error[=*file_name*]

  出错日志的文件名。如果没有给出 *file_name*，则以数据目录里的 *HOSTNAME*.err 为默认的日志文件名，其中 *HOSTNAME* 是服务器主机的名字。如果给出的 *file_name* 是一个相对路径名，则以数据目录为起点解释它。如果给出的 *file_name* 没有扩展名，mysqld 将给它加上“.err”作为扩展名。

- --log-isam[=*file_name*]

  启用索引文件日志机制。这只用于对 MyISAM 操作进行调试的场合。如果你没有给定 *file_name* 参数，默认日志文件名将是数据目录中的 myisam.log。

- --log-long-format

把辅助性信息写到二进制日志和慢查询日志里去。这个选项已被弃用。辅助性信息是默认记录的，可以用--log-short-format 禁用。

- --log-output[=*destinations*]

  这个选项为常规查询日志和慢查询日志指定一个日志信息输出地点，如果相应的日志功能已启用的话。*destinations* 是一个由一个或多个以逗号分隔的地点名构成的列表，地点名的可取值是 TABLE、FILE 和 NONE。其中 NONE 的优先级最高，它将禁用日志功能。如果完全省略这个选项或是省略了这个选项的设置值，则以 FILE 为默认设置（MySQL 5.1.6 到 5.1.20 版以 TABLE 为默认设置）。

  --log [= *file_name*]选项将启用常规日志功能，并根据你是否给出了可选的 *file_name* 值把日志信息写到本选项指定的输出地点或指定的日志文件。--general-log 或 --skip-general-log 选项则是只启用或禁用通用日志功能，不指定日志文件。--log-slow-queries、--slow-query-log 和--skip-slow-query-log 选项有着类似的效果，但针对的是慢查询日志。

  general_log 或 slow_qurey_log 系统变量可以实时设置以启用或禁用相应的日志功能。general_log_file 或 slow_qurey_log 系统变量可以实时设置以改变相应的日志文件的名字。

  这个选项是从 MySQL 5.1.6 版开始引入的。在那之前，日志信息总是写入一个文件。

- --log-queries-not-using-indexs（布尔）

  如果已经启用了慢查询日志功能，这个选项将把没有使用索引的查询也记载到慢查询日志里去。

- --log-short-format（布尔）

  把较少的信息写入二进制日志和慢查询日志，如果那些日志已被激活的话。

- --log-slow-admin-statements（布尔）

  诸如 ALTER TABLE 或 OPTIMIZE TABLE 语句之类的系统管理操作有可能很慢，但它们在默认的情况下并不会被记载到慢查询日志里。这个选项将导致它们当中比较慢的也被记入日志。它是从 MySQL 5.0.8 版开始引入的。

- --log-slow-queries[=*file_name*]

  从 MySQL 5.1.6 版开始，这个选项将启用慢查询日志功能，并根据你是否给出了可选的 *file_name* 值把日志信息写到--log-output 选项指定的输出地点或是指定的日志文件。在 5.1.6 版之前，这个选项将把日志信息写入一个文件。如果没有给出 *file_name* 值，默认的日志文件名将是数据目录里的 *HOATNAME*.-slow.log，其中 *HOATNAME* 是服务器主机的名字。如果给出的 *file_name* 值是一个相对路径名，则以数据目录为起点来解释。

- --log-tc=*file_name*

  事务协调器日志文件（用于 XA 事务）的路径名。这个选项目前尚未正式投入使用。它是从 MySQL 5.0.3 版开始引入的。

- --log-tc-size=*n*

  事务协调器日志文件的长度。这个选项是从 MySQL 5.0.3 版开始引入的。

- --log-update[=*file_name*]

  启用与变更日志有关的日志机制。这个选项已弃用，变更日志自 My SQL 5.0 已删除，所以如果你没给出--log-bin，--log-update 将启用二进制日志。

- --log-warnings[=*n*]或-W[*n*]

将某些非关键性警告信息写入错误日志内。这个选项是默认启用的，给出这个选项时不设置值也能启用警告。如果设置为 0 或者 1，将分别禁用或启用警告。如果指定两次此选项，或者将其值设置为 2，将启用已中止连接（从 MySQL 5.2.6 起）或“拒绝访问”错误的日志功能。如果 *n* 是用`-W`设置的，它们之间不能有空格，否则 *n* 将不能得到正确解释。

- `--low-priority-updates`（布尔）
  给修改操作分配比检索操作更低的优先级。
- `--memlock`（布尔）
  如果可能，锁定内存中的服务器。这个选项只在 Solaris 这种系统上起作用，且要求 MySQL 服务器必须运行为`root`用户。
- `--myisam-recover[=level]`
  启用 MyISAM 数据表的自动恢复机制。当服务器打开一个 MyISAM 表时，如果这个表标记为已崩溃或是上次使用后没有正确关闭，就会进行一次修复。`level`可以为空，即禁用这种恢复机制；也可以是以逗号分隔的一个或多个选项值：`DEFAULT`（直接恢复，不做任何其他特殊处理，与没给出任何选项的情况等价），`BACKUP`（给修改过的数据表创建一个备份），`FORCE`（即使可能造成多个数据行丢失，也要强行进行恢复），`QUICK`（快速恢复）。
  如果采用了`--delay-key-write`选项来运行服务器，或者 MyISAM 表已配置为启用延迟索引写操作，这个选项就很有用。
- `--ndbcluster`（布尔）
  启用 NDBCLUSTER 存储引擎。如果已经编译有 NDBCLUSTER 功能，该引擎将默认启用。如果根本用不着 NDBCLUSTER 数据表，建议使用`--skip-ndbcluster`选项禁用 NDBCLUSTER 存储引擎以节约内存。
- `--new`或`-n`
  使用新的、但可能不太完善的例程。它们是 MySQL 软件中稳定性尚未得到最终肯定的试验性功能。如果你不想冒险，那就最好不要使用这个选项。
- `--old`（布尔）
  设置`old`系统变量，这将启用某些功能的早期行为。这个选项是从 MySQL 5.1.8 版开始引入的。
- `--old-passwords`
  MySQL 4.1 及以后的版本使用了一种更安全的口令加密方法。口令仍按老方法加密的账户在新版本里仍能使用，但新口令却都是用新方法加密的。这个选项将强制服务器使用老方法来加密新口令。（如果你想让新版服务器支持老口令或者想把账户转移到老版本的服务器上，就需要使用这个选项。）
- `--old-style-user-limits`（布尔）
  MySQL 系统管理员可以对各 MySQL 账户的资源耗用量进行限制，关于这方面的讨论见 12.3.1 节的第 5 小节。在 MySQL 5.0.3 版之前，如果某个账户可以从多个主机连接服务器，其资源耗用量将按主机的不同而分别计算。从 MySQL 5.0.3 版开始，在判断给定账户的资源耗用量是否超限时不再区分该账户是从哪个主机连接的，但`--old-style-user-limits`选项可以用来启用早期 MySQL 版本所使用的资源耗用量评估方法。
- `--one-thread`
  只使用一个线程来运行服务器，用于 Linux 系统上的调试工作（Linux 系统通常至少使用 3 个线程）。这个选项从 MySQL 5.1.17 版本开始已弃用，被`--thread_handling=one-thread`取代。

- --pid-file=file_*name*
  在启动的时候，mysqld 会把自己的进程 ID（process ID，即 PID）写到一个文件里。这个选项给出的就是那个 PID 文件的路径名。其他进程可以通过这个文件来确定服务器的进程 ID——通常是为了向它发出一个信号。比如说，当 mysql.server 脚本需要向服务器发出一个关机信号时，它就会先去读取这个文件。如果 *file_name* 是一个相对路径，mysqld 将以数据目录为起点来解释它。这个选项在嵌入式服务器上没有效果。
- --safe-mode
  这个选项类似于--skip-new 选项，但将禁用更多的功能。如果 MySQL 运行得不太稳定，或者复杂查询的结果看起来不太正确，你就应该试试这个选项。
- --safe-show-database（布尔）
  这个选项已被淘汰。MySQL 管理员应该使用 SHOW DATABASES 权限来管理对数据库名的访问。
- --safe-user-create（布尔）
  如果某用户不具备 user 权限数据表的 INSERT 权限，则不允许该用户创建新的账户。
- --safemalloc-mem-limit=*n*
  模拟内存短缺的情况。这个选项给可分配内存量设定了一个上限，它只能用在服务器在编译阶段的配置工作中使用了--with-defug=full 选项的场合。
- --secure-auth（布尔）
  要求客户必须使用从 MySQL 4.1 系列版本开始启用的更安全的口令格式，否则不允许连接。
- --secure-file-priv=*dir_name*
  设置 secure-file-priv 系统变量以限制对给定目录进行的某些文件操作。这个选项是从 MySQL 5.0.38/5.1.17 版开始引入的。
- --skip-grant-tables（布尔）
  不使用权限表来验证客户连接。这将允许任何客户去做任何事情。它还将禁用 CREATE USER、DROP USER、RENAME USER、GRANT、REVOKE 以及 SET PASSWORD 语句。如果想让服务器开始使用权限表来验证客户连接，可以发出一条 FLUSH PRIVILEGES 语句或者发出一条 mysqladmin flush-privileges 命令，或者重启服务器且不用--skip-grant-tables。
- --skip-host-cache
  禁止使用主机名缓存。
- --skip-locking
  这个选项已弃用，被--skip-external-locking 取代。参见对--external-locking 选项的介绍。
- --skip-name-resolve
  不对主机名进行解析。如果使用了这个选项，就必须在权限表里把主机名写成数字形式的 IP 地址或者 localhost。
- --skip-networking
  不允许 TCP/IP 连接，只允许本地客户连接服务器，并且必须使用非 TCP/IP 接口连接。Unix 客户可以使用 Unix 套接字文件连接。Windows 客户可以使用共享内存或给定管道连接。
- --skip-new
  不使用新的、但可能不太完善的例程。
- --skip-safemalloc
  不进行内存分配检查。这个选项只能用在 MySQL 服务器在编译阶段的配置工作中使用了

--with-defug=full 选项的场合。

- --skip-show-database
  默认情况下，任何用户都可以发出 SHOW DATABASES 语句，这个语句将显示出该用户具有 SHOW DATABASES 权限的所有数据库，以及用户具有其他某些权限的数据库。如果采用--skip-show-database 选项，就只有具备 SHOW DATABASES 权限的用户才能使用 SHOW DATABASES 语句，这时将显示所有数据库。
- --skip-stack-trace
  在执行出错时不打印栈跟踪信息。
- --skip-symlink
  这个选项已弃用，由--skip-symbolic-links 取代。参见--symbolic-link 条目。
- --skip-thread-priority
  在普通情况下，数据修改操作（会改变数据表内容的查询）的优先级要高于数据检索操作。如果这不是你所希望的，就需要使用这个选项让服务器不给不同类型的查询赋予不同的优先级。
- --slow-query-log（布尔）
  启用慢查询日志功能并把日志信息写到--log-output 选项指定的输出地点。--skip--slow-query-log 选项将禁用该日志。这个选项是从 MySQL 5.1.12 版开始引入的。
- --sql-bin-update-same（布尔）
  把 sql-log-bin 和 sql-log-update 系统变量合二为一，只需（使用 SET 语句）设置其中之一就可以把另一个也设置好。这个选项从 MySQL 5.0 系列版本开始被淘汰，这是因为更新日志已不复存在，与更新操作有关的日志信息将写入二进制日志。
- --sql-mode=*mode_list*
  这个选项的用途是改变服务器的某些特定行为，让它更符合 SQL 语言标准，或是让它与其他数据库系统或早期 MySQL 服务器更好地兼容。*mode_list* 是一个由一个或多个以逗号分隔的模式值构成的列表，如果该列表是一个空字符串，其含义是清除此前的所有设置。可供选用的模式值见附录 D 里关于 sql-mode 系统变量的描述。
- --symbolic-links（布尔）
  在 Unix 系统上，这个选项将允许使用 MyISAM 数据表的数据文件和索引文件的符号链接（比如用在 DATA DIRECTORY 和 INDEX DIRECTORY 数据表创建选项里）。在 Windows 系统上，这个选项将允许使用数据库目录的符号链接。关于这些技巧的讨论见第 11 章。Windows 系统上的数据库符号链接支持是默认启用的，可以用--skip-symbolic-links 选项禁用。
- --sync-frm（布尔）
  让服务器在创建每个.frm 文件的同时把它同步创建在硬盘上。这个选项是默认启用的，你可以用--skip-sync-frm 选项禁用。
- --sysdate-is-now（布尔）
  从 MySQL 5.0.13 版开始，SYSDATE()函数的返回值是该函数被调用时的日期和时间，NOW()函数的返回值则是该函数所在的语句开始执行时的时间。
  --sysdate-is-now 选项将使 SYSDATE()函数的行为和 NOW()函数相似。这个选项是从 MySQL 5.0.20 版开始引入的。
- --tc-heuristic-recover=*str*
  因为发生了两阶段提交而导致崩溃时将采用的恢复策略。可供选用的设置值有 COMMIT 或

ROLLBACK。这个选项目前尚未正式投入使用，它是从 MySQL 5.0.3 版开始引入的。

- --temp-pool（布尔）

  如果使用了这个选项，服务器就将使用一组数量有限的名字来命名临时文件，而不是为每个临时文件创建一个独一无二的名字。在 Linux 系统上，这种做法能避免某些缓存问题。这个选项是默认启用的，使用--skip-temp-pool 可以禁用它。

- --timed_mutexes（布尔）

  让服务器收集与 InnoDB 互斥锁定机制有关的时间设置信息。这个选项是从 MySQL 5.0.3 版开始引入的。

- --transaction-isolation=*level*

  设定默认的事务隔离级别。参数 level 的可取值包括 READ-UNCOMMITTED、READ-COMMITTED、REPEATABLE-READ 和 SERIALIZABLE。

- --tmpdir=*dir_name* 或 -t *dir_name*

  用来存放临时文件的目录的路径名。这个选项的值允许是一组子目录，MySQL 将以轮转方式来使用这些子目录。在 Unix 系统上，子目录名之间要用冒号来分隔，在 Windows 或 NetWare 系统上要用分号来分隔。

- --warnings[=*n*]

  已弃用，由--log-warnings 取代。

1. Windows选项

本节中的选项只能在运行于 Windows 下的服务器上使用。服务名和命名管道名称不区分大小写，共享内存名称区分大小写。

- --console（布尔）

  用一个控制台窗口来显示出错信息。

- --enable-named-pipe（布尔）

  对于包含命名管道支持的 MySQL 服务器，命名管道连接是默认禁用的。这个选项将启用命名管道连接。默认的管道名是 MySQL。这个名字可以用--socket 选项来更改。

- --install[*service_name*]

  把服务器安装为一项服务并让它在 Windows 启动时自动运行。如果没给出 *service_name*，默认服务名就是 MySQL。

- --install-manual[*service_name*]

  把服务器安装为一项服务但不让它在 Windows 启动时自动运行，你必须明确地以手动方式启动。如果没有给出 *service_name*，默认服务名就是 MySQL。

- --remove[*service_name*]

  删除作为一项服务的服务器。如果没有给出 *service_name*，默认服务名就是 MySQL。

- --shared-memory（布尔）

  启用对以共享内存方式建立的连接的支持。默认的共享内存名是 MYSQL，这个名字可以通过--shared-memory-base-name 选项来改变。

- --standalone

  把服务器运行为一个独立的程序而不是一项服务。

2. InnoDB选项

本小节中的选项都是 InnoDB 存储引擎所独有的。

- `--innodb`（布尔）

  启用 InnoDB 存储引擎。如果 MySQL 服务器在编译时包括了对 InnoDB 存储引擎的支持功能，该引擎将默认启用。如果根本用不着 InnoDB 数据表，可以用`--skip-innodb` 选项禁用 InnoDB 存储引擎以节约内存。
- `-- innodb_autoextend_increment=`*`size`*

  如果已经把 InnoDB 共享表空间配置成可自动扩展的，你可以用这个选项来控制该表空间在每次扩展时的长度增量。这个选项的值以兆字节为单位，其默认值是 8 MB。
- `--innodb_data_file_path=`*`filespec_list`*

  InnoDB 表空间组件文件的定义。*`filespec_list`* 的格式在 12.6.3 节的第 1 小节里有详细的介绍。
- `--innodb_data_home_dir=`*`dir_name`*

  用来存放 InnoDB 表空间组件文件的子目录的路径名。
- `--innodb_fast_shutdown`（布尔）

  加快服务器的关机过程。InnoDB 存储引擎将跳过它平时要进行的一些操作。
- `--innodb_file_per_table`（布尔）

  如果启用了这个选项，InnoDB 存储引擎将为每个新数据表单独创建一个表空间文件，也就是让每一个 InnoDB 数据表在它的数据库子目录里都有一个对应的.ibd 文件。此时，InnoDB 共享表空间只用来存放 InnoDB 数据字典项，不再用来存放数据和索引。这个选项是默认禁用的。
- `--innodb_flush_log_at_trx_commit=`*`n`*

  这个选项的默认值是 1，其含义是在你提交事务时清空 InnoDB 日志，这保证了 `ACID` 属性。把这个选项设置为 0 将减少 InnoDB 对磁盘的写操作次数，但同时却加大了系统崩溃时丢失一些最近被提交的事务的可能性。`n` 的可取值如表 F-11 所示。

表 F-11

取　值	含　义
0	每隔一秒写一次日志，同时刷新相应的磁盘文件
1	每提交一次事务写一次日志，同时刷新相应的磁盘文件
2	每提交一次事务写一次日志，但每隔一秒刷新一次相应的磁盘文件

- `--innodb_log_arch_dir=`*`dir_name`*

  这个选项尚未使用已从 MySQL5.1.21 中删除。
- `--innodb_log_archive=`*`n`*

  这个选项目前尚未使用。
- `--innodb_log_group_home_dir=`*`dir_name`*

  用来存放 InnoDB 日志文件的子目录的路径名。
- `--innodb_max_dirty_pages_pct=`*`n`*

  当 InnoDB 缓冲区池里的“脏”（意思是数据被修改过）页面达到百分之 *`n`* 时，InnoDB 存储引擎就会把日志信息及时“冲洗”到硬盘。这个选项的可取值是从 0 到 100，默认值是 90。
- `--innodb_safe_binlog`

  在 InnoDB 存储引擎进行崩溃恢复工作之后，把二进制日志截短到无法回滚的最后一个语句或事务。

- --innodb_status_file（布尔）

  定期把 SHOW INNODB STATUS 语句的输出信息写到数据目录里名为 innodb_status.*nnnnnn* 的文件里，*nnnnnn* 是服务器进程的 ID 号。这些状态文件在服务器下一次正常关机之前不会被删除，所以应该定期对它们当中不再有保留价值的进行清理。

3. 与复制机制有关的选项

本小节中的选项都是 MySQL 的复制机制所独有的。

有几个名字形式为--master-xxx 的复制机制选项没有列在这里。它们是用在从服务器上的，用于指定连接主服务器的参数，从 MySQL 5.1 开始已经弃用，从 MySQL 5.2 中被删除。现在你可以用 CHANGE MASTER 语句来指定参数。

--report-xxx 和--show-slave-auth-info 信息影响着主服务器上 SHOW SLAVE HOSTS 的输出，如附录 E 所述。

- --abort-salve-event-count=*n*

  这个选项是 MySQL 用来测试复制机制的工作情况。

- --binlog-do-db=*db_name*

  供主服务器使用，含义是只记录指定数据库上的数据修改操作，其他数据库将不参加复制机制。如果想记录多个数据库上的数据修改操作，就必须使用多个--binlog-do-db 选项来依次指定每一个数据库。

- --binlog-ignore-db=*db_name*

  供主服务器使用，含义是不记录指定数据库上的数据修改操作。如果想忽略多个数据库上的数据修改操作，就必须使用多个--binlog-ignore-db 选项来依次指定每一个数据库。——注意，这个选项会让二进制日志不能包含发生崩溃时恢复给定数据库需要用到的信息。为避免这个问题，可在从服务器上以--replication-ignore-db 代替它。

- --disconnect-slave-event-count=*n*

  这个选项由 MySQL 测试套件用来测试复制机制的工作情况。

- --init-rpl-role=*val*

  表明这个服务器在复制机制中的角色，val 的可取值是 master 或 salve。这个选项由 MySQL 测试套件用来测试复制机制的工作情况。

- --init-slave=*str*

  这个选项应该在复制机制中的主服务器上使用，它给出的语句将在每个从服务器连接该主服务器时执行。这个选项的值应该是一个或多个以分号隔开的 SQL 语句。

- --log-slave-updates（布尔）

  这个选项将使从服务器把来自主服务器的变更情况记录到它自己的二进制日志。这个选项使这个从服务器还能够充当另一个从服务器的主服务器，这样你就能够用多个服务器构成一个复制链。

- --master-info-file=*file_name*

  供从服务器使用，指定保存当前复制状态信息的文件的名字。这个文件的内容包括：复制机制中主二进制日志的文件名和位置、主服务器所在的主机、用户名、口令、端口号，以及连接重试间隔时间和 SSL 选项值等。这个文件的默认名是数据目录里的 master.info 文件。如果 file_name 是以相对路径给出的，程序将以数据目录为起点解释它。

- --master-retry-count=*n*

  供复制机制中的从服务器使用，如果在经过 *n* 次重试之后仍未成功连接上主服务器，则放弃

本次操作。

- `--max-binlog-dump-events=`*n*
  这个选项由 MySQL 测试套件用来测试复制机制的工作情况。
- `--relay-log=`*file_name*
  供从服务器使用，负责给定中继日志的文件名。（在 MySQL 4 里，从服务器的 I/O 线程负责把从主服务器那里读到的数据变更情况写到中继日志，再由 SQL 线程负责从中继日志里读出语句并完成相应的操作。这个文件的默认名是数据目录里的 `HOSTNAME-relay-bin.`*nnnnnn*，其中的 *HOSTNAME* 是该服务器主机的主机名，*nnnnnn* 则是一个按 1 递增的序号（每创建一个新日志，*nnnnnn* 的值就增加 1）。
- `--relay-log-index=`*file_name*
  供从服务器使用，负责给定中继日志索引文件的名字。这个文件的默认名是数据目录里的 *HOSTNAME*`-relay-bin.index`，其中 *HOSTNAME* 是该服务器主机的主机名。如果 *file_name* 是按相对路径给出的，程序将以数据目录为起点作出解释。
- `--relay-log-info-file=`*file_name*
  供从服务器使用，负责给定中继日志信息文件的文件名。这个文件的默认名是数据目录里的 `relay-log.info`。
- `--replicate-do-db=`*db_name*
  供从服务器使用，含义是只复制指定数据库上的修改操作。如果想处理多个数据库上的修改操作，就必须使用多个`--replicate-do-db` 选项来依次指定每一个数据库。
- `--replicate-do-table=db_name.`*tbl_name*
  供从服务器使用，含义是只复制名字格式为 *db_name.tbl_name* 的指定数据表上的修改操作。如果想复制多个数据表上的修改操作，必须使用多个`--replicate-do-table` 选项来依次指定每一个数据表。
- `--replicate-ignore-db=`*db_name*
  供从服务器使用，含义是不复制指定数据库上的修改操作。如果想忽略多个数据库，就必须使用多个`--replicate-ignore-db` 选项来依次指定每一个数据库。
- `--replicate-ignore-table=`*db_name.tbl_name*
  告诉从服务器不要复制给定的数据表。如果需要指定多个数据表，必须为每个数据表分别写出一个`--replicate-ignore-table` 选项。
- `--replicate-rewrite-db=`*maaster_db->slave_db*
  告诉从服务器要把某个数据库当做另外一个数据库对待。对主服务器上的 *master_db* 数据库的更改将复制到从服务器中的 *slave_db* 上。这个选项只在 *master_db* 为默认服务器时才能使用，而且只能用于在那个数据库的表上操作的语句。在命令行上给出这个选项时，其值应用引号引述，防止命令解释器将“>”字符当做一个输出重定向操作符。这个选项可以多次给出，服务器将按顺序使用它们，而且采用 *master_db* 值匹配到的第一个规则。
  这个选项在测试其他`--replication-`*xxx* 选项指定的操作之前应用，所以如果你使用了它，那些选项应该把 *slave_db* 用作数据库名。
- `--replicate-same-server-id`（布尔）
  如果启用了这个选项，服务器将不会跳过包含它本身的服务器 ID 的复制事件。为避免发生复制循环现象，这个选项是默认禁用的，但在某些特殊的场合有可能需要激活它。

- ❑ --replicate-wild-do-table=*pattern*
  供从服务器使用，含义是只对名字与给定模式相匹配的数据表进行复制处理。如果想使用多个匹配模式，就必须使用多个--replicate-wild-do-table 选项来依次给出每一个模式。
- ❑ --replicate-wild-ignore-table=*pattern*
  供从服务器使用，含义是不对名字与给定模式匹配的数据表进行复制处理。如果想使用多个匹配模式，就必须使用多个--replicate-wild-ignore-table 选项来依次给出每一个模式。
- ❑ --report-host=*host_name*
  向主服务器报告这个从服务器的主机名 *host_name*。
- ❑ --report-password=*pass_val*
  向主服务器报告这个从服务器的账户口令 *pass_val*。
- ❑ --report-port=*port_num*
  向主服务器报告这个从服务器的端口号 *port_num*。
- ❑ --report-user=*user_name*
  向主服务器报告这个从服务器的账户名 *user_name*。
- ❑ --rpl-recovery-rank=*n*
  这个选项未正式投入使用。
- ❑ --server-id=*n*
  复制机制中的主服务器的 ID 值。这个值必须是 1 到 $2^{32}-1$ 之间的某个数，并且在复制机制所涉及的各个服务器中必须是独一无二的。
- ❑ --show-slave-auth-info（布尔）
  让主服务器在 SHOW SLAVE STATUS 语句的输出报告里显示从服务器的用户名和口令。
- ❑ --skip-slave-start
  不自动启动从服务器线程，必须使用 START SLAVE 语句以手动方式来启动它。
- ❑ --salve-allow-batching（布尔）
  设置 salve-allow-batching 系统变量。这个选项是从 MySQL 5.2.5 版开始引入的。
- ❑ --slave-load-tmpdir=*dir_name*
  从服务器用来处理 LOAD DATA 语句的子目录的路径名。若未指定这个选项，其默认值将为--tmpdir。
- ❑ --slave-skip-errors=*error_list*
  如果在执行过程中发生了 *error_list* 里列出的错误，从服务器将忽略之而不是挂起复制处理进程。（但通常最好找出问题的根源，这样才能解决它们，而不必使用这个选项来忽略它们。）如果 *error_list* 参数的值是 all，则将忽略所有错误；否则，*error_list* 参数的值就应该是一个或者多个以逗号分隔的出错代码。
- ❑ --sporadic-binlog-dump-fail（布尔）
  这个选项由 MySQL 测试套件用来测试复制机制的工作情况。

F.12.3　与**mysqld**有关的变量

下面这条命令能让你查看到各 mysqld 系统变量的默认值：

```
% mysqld --verbose --help
```

下面这条命令能让你查看到 mysqld 程序当前正使用的系统变量及其取值：

```
% mysqladmin variables
```

mysqld 系统变量的当前值还可以用 SHOW VARIABLES 语句来查看，这些变量可以在附录 D 中查到。系统变量都可以在启动时按 F.2.1 节的第 2 小节所介绍的步骤设置。此外，许多系统变量还可以进行动态设置，这方面的详细情况请参见 12.5.1 节和附录 E 里的 SET 语句条目。

F.13 mysqld_multi

mysqld_multi 脚本大大简化了在同一台主机上运行多个 mysqld 服务器的工作，你既可以用它来启动或停止服务器，也可以用它来查看运行情况。

mysqld_multi [*options*] *command server_list*

command 参数的值应该是 start、stop 或 report，server_list 参数值是你打算操作的服务器的名字。mysqld_multi 脚本的具体用法请参见 12.11.4 节。

F.13.1 mysqld_multi支持的标准选项

```
--help          --silent        --verbose
--password      --user          --version
```

--silent 和--verbose 从 MySQL 5.0.2 开始可以使用。

当你使用 mysqld_multi 脚本来停止服务器或者查看它是否在运行时，mysqld_multi 脚本将把--user 和--password 选项值传递给 mysqladmin 程序。

F.13.2 mysqld_multi独有的选项

- --config-file=*file_name*
  从 MySQL 5.0.42/5.1.18 起，这个选项已弃用，由标准的--defaults-extra-file 选项取代。对于早期版本，这是一个选项文件名，mysqld_multi 脚本将使用其中的选项来操纵服务器。如果没有使用--config-file 选项，mysqld_multi 脚本将读/etc/my.cnf 文件和登录目录中的.my.cnf 文件以获得服务器选项。（mysqld_multi 脚本会到标准选项文件里读取它自己的选项，--config-file 选项不改变这一行为。）
- --example
  显示一份选项文件样本，它演示了适用于 mysqld_multi 脚本的各种选项文件组的用法。
- --log=*file_name*
  mysqld_multi 脚本用来记录其操作情况的日志文件的名字。如果这个文件已经存在，新日志信息将被追加到这个文件的末尾。默认的日志文件是数据目录中的 mysqld_multi.log。你可以用--no-log 选项禁用此功能。
- --mysqladmin=*file_name*
  你要使用的 mysqladmin 程序所在的子目录的路径名。如果 mysqld_multi 脚本无法找到 mysqladmin 程序，或者你想让它使用某个特定版本的 mysqladmin 程序，就需要使用这个选项。
- --mysqld=*file_name*
  你要使用的 mysqld 程序所在的子目录的路径名。如果 mysqld_multi 脚本无法找到 mysqld 程序，或者你想让它使用某个特定版本的 mysqld 程序，就需要使用这个选项。你也可以用这个选项来给出 mysqld_safe 或 mysqld 的路径名。

- `--no-log`
  显示日志输出而不是把它写到日志文件里去。如果想看屏幕上的，日志输出内容，就必须使用这个选项，因为默认行为是把它们写到日志文件。
- `--tcp-ip`
  在默认情况下，mysqld_multi 脚本将试图使用一个 Unix 套接字文件来连接服务器。这个选项将使用 TCP/IP 去建立这种连接。如果某个正在运行的服务器的套接字文件已经被删除，你就只能通过 TCP/IP 去访问它，而这就需要你使用这个选项。

F.14 mysqld_safe

mysqld_safe 用来启动和监控 mysqld 服务器：

```
mysqld_safe [options]
```

如果服务器意外“死亡”，mysqld_safe 将重新启动它。mysqld_safe 是一个 shell 脚本，在 Unix 上可用。NetWare 上也有一个已编译版本。

F.14.1 mysqld_Safe支持的标准选项

`--help`

--help 从 MySQL 5.0.3 开始可用。

F.14.2 mysqld_safe独有的选项

能够与mysqld程序一起使用的选项都可以与mysqld_safe脚本一起使用——后者将把这些选项传递给前者。此外，mysqld_safe 脚本还有以下一些独有的选项：

- `--basedir=`*dir_name*
  MySQL 安装目录的路径名。
- `--core-file-size=`*n*
  把服务器发生崩溃时生成的 core 文件长度限制为 *n* 个字节。
- `--datadir=`*dir_name*
  MySQL 数据目录的路径名。
- `--err-log=`*file_name*
  这是--log-error 的旧形式。
- `--ledir=`*dir_name*
  mysqld_safe 脚本将到这个子目录（即所谓的“libexec”子目录）里寻找服务器。
- `--log-error=`*file_name*
  用来保存出错日志信息的文件名。相对路径名将以你执行 mysqld_safe 脚本时所在的子目录为起点进行解析。如果根本没有给出这个选项，默认的出错日志文件名将是数据目录里的 *HOSTNAME*.err，其中 *HOSTNAME* 是当前主机的名字。
- `--mysqld=`*file_name*
  mysqld 程序的路径名。
- `--mysqld-version=`*suffix*
  这个选项的值是一个后缀字符串。如果给出了这个选项，mysqld-safe 脚本将先用一个连字符把该后缀追加到基本名 mysqld 的末尾，然后再使用如此拼凑出来的名字去启动指定的服务器。

- --open-files=*n*、--open-files-limit=*n*
  mysqld 程序应该保留的文件描述符的个数。
- --pid-file=*file_name*
  mysqld 程序的进程 ID 文件的名字。
- --port=*port_num*
  让 MySQL 服务器在指定端口上监听 TCP/IP 连接。
- --port-open-timeout=*n*
  告诉 MySQL 服务器在启动时应该等待 *n* 秒以后再去检查它的 TCP/IP 端口是否可用。
- --skip-kill-mysqld
  在启动一个新的 mysqld 进程之前，不要尝试“杀死”任何当前正在运行的 mysqld 进程。如果想运行同一个 mysqld 程序的多个实例的话，这个选项会很有用。这个选项只在 Linux 系统上有效。
- --skip-syslog
  让服务器不要把出错消息发送到 syslog，把它们发送到一个日志文件。这个选项的默认设置是把出错消息发送到一个日志文件。这个选项是从 MySQL 5.1.20 版开始引入的。
- --socket=*file_name*
  Unix 套接字文件的路径名。
- --syslog
  让服务器把出错消息发送到 syslog——如果系统里有 logger 程序正在运行的话。这个选项是从 MySQL 5.1.20 版开始引入的。
- --syslog-tag=*tag*
  在把出错消息发送到 sylog 时，来自 mysqld-safe 脚本和 mysqld 程序的消息都带有一个相应的程序名标记作为前缀。--syslog-tag 选项将把这样的标签分别修改成 mysqld-safe-*tag* 和 mysqld-*tag*。这个选项是从 MySQL 5.1.21 版开始引入的。
- --timezone=*tz_name*
  把服务器的系统时区设置为 *tz_name*。如果服务器无法自动确定系统时区，这个选项将很有用。
- --user=user_name、--user=*uid*
  用来运行 MySQL 服务器的系统账户的用户名或者数值形式的用户 ID。

F.15 mysqldump

mysqldump 程序能够把数据表的内容写到文本文件中。它生成的那些文件有许多用途：比如作为数据库备份、把数据库转移到另一个服务器去、根据现有数据库的内容建立一个供测试工作使用的数据库，等等。

在默认情况下，用 mysqldump 程序导出的数据表将输出为一个文件，这个文件由一条 CREATE TABLE 语句（用来创建该数据表）和紧随其后的一组 INSERT 语句（用来加载该数据表内容的）构成。但如果你使用了--tab 选项，mysqldump 程序将生成两个文件：数据表的内容将以纯数据格式被写到一个数据文件里，每个数据记录一行，而用来创建该数据表的 SQL 语句将被写到另外一个文件里。

mysqldump 程序有以下 3 种运行模式：

```
mysqldump [options] db_name [tbl_name] ...
mysqldump [options] --databases db_name ...
```

```
mysqldump [options] --all-databases
```

在第一种模式里，mysqldump 程序将转储指定数据库里的指定数据表。如果没有给出数据表的名字，mysqldump 将依次转储该数据库里的所有数据表。在第二种模式里，mysqldump 程序将把所有的参数都解释为数据库的名字，它将依次转储每个数据库里的所有数据表。在第三种模式里，mysqldump 程序将转储所有数据库里的所有数据表。如果使用了--database 或--all-database，输出将包含 CREATE DATABASE IF NOT EXISTS 和 USE 语句，它们位于每个数据库数据表的语句之前。

下面是 mysqldump 程序最常见的使用方法：

```
% mysqldump db_name > backup_file
```

注意：用 mysqldump 程序导出的备份文件是不能用 mysqlimport 程序重新导入到 MySQL 里的，只能使用 mysql 程序来导入，如下所示：

```
% mysql db_name < backup_file
```

mysqldump 忽略掉 INFORMATION_SCHEMA 数据库不转储，即使在命令行上显式指定了它。

F.15.1　mysqldump支持的标准选项

```
--character-sets-dir      --password                --socket
--compress                --pipe                    --user
--debug                   --port                    --verbose
--default-character-set   --protocol                --version
--help                    --set-variable
--debug-check             --host
--debug-info              --shared-memory-base-name
```

--debug-info 和--debug-check 选项分别是从 MySQL 5.0.32/5.1.14 版本和 MySQL 5.1.21 开始新增加。4 开始，mysqldump 程序还支持各种标准的 SSL 选项。

F.15.2　mysqldump独有的选项

下面介绍的各个选项都是用来控制 mysqldump 程序的操作行为的。在 F.15.3 节里，我们将对那些用来配合--tab 选项以决定数据文件格式的选项做集中的介绍。

- --add-drop-database（布尔）
  在每次执行 CREATE TABLE 语句之前先执行一条 DROP TABLE IF EXIST 语句。这个选项是从 MySQL 5.0.7 版开始引入的。
- --add-drop-table（布尔）
  在每条 CREATE TABLE 语句的前面加上一条 DROP TABLE IF EXIST 语句。
- --add-locks（布尔）
  在用来加载各数据表内容的各组 INSERT 语句的前后分别加上 LOCK TABLE 和 UNLOCK TABLE 语句。
- --all 或 -a（布尔）
  已弃用，被--create-options 代替。
- --all-databases 或 -A（布尔）
  转储所有数据库中的所有数据表。这个选项也使转储输出包含 CREATE DATABASE 和 USE 语句。
- --all-tablesapce 或-Y（布尔）
  转储所有的表空间。这个选项是从 MySQL 5.1.6 版开始引入的。

- --allow-keywords（布尔）
  允许创建以关键字作为名字的数据列。
- --apply-slave-statements（布尔）
  这个选项需要与--dump-salve 选项配合使用。它将在 mysqldump 程序的输出结果里在每条 CHANGE MASTER 语句的前、后分别加上一条 STOP SLAVE 语句和一条 START SLAVE 语句。这个选项是从 MySQL 6.0.4 版开始引入的。
- --comments 或-i（布尔）
  在输出结果里增加一些提示性注释，如 mysqldump 程序的版本号、每组 INSERT 语句作用于哪个数据表，等等。这个选项是默认启用的，你可以用--skip-comments 选项来禁用。
- --compact（布尔）
  生成比较简洁的输出，不包含注释，对设置系统变量的语句也不做解释。这个选项还将同时启用--skip-add-drop-table、--skip-set-charset、--skip-disable-keys 和--skip-add-locks 选项。
- --compatible=*mode*
  这个选项将导致 mysqldump 程序对其输出结果进行必要的修改，以便与 SQL 语言标准、其他数据库系统或 MySQL 服务器的早期版本保持兼容。*mode* 值用来设定兼容模式，你可以用一个以逗号为分隔符的列表来给出多个如表 F-12 所示的模式值：

表 F-12

选 项	兼容性含义
ANSI	与ANSI标准兼容
DB2	与DB2兼容
MAXDB	与MaxDB兼容
MSSQL	与MS SQL Server兼容
MYSQL323	与MySQL 3.23兼容
MYSQL40	与MySQL 4.0兼容
ORACLE	与OPACLE兼容
POSTRESQL	与PostgreSQL兼容
NO_FIELD_OPTIONS	清除MySQL独有的与数据列有关的选项
NO_KEY_OPTIONS	清除MySQL独有的与索引有关的选项
NO_TABLE_OPTIONS	清除MySQL独有的与数据表有关的选项

  如果 mysqldump 程序连接的是一个 4.0.1 版之前的 MySQL 服务器，这个选项将没有任何效果。
- --complete-insert 或-c（布尔）
  在生成 INSERT 语句时，把将被插入的每一个数据列都写出来。
- --create-options（布尔）
  在 mysqldump 程序生成的 CREATE TABLE 语句里增加一些注释性信息，如存储引擎、AUTO_INCREMENT 序列的起始值等。这些信息可以在 CREATE TABLE 语句里直接用作相应的 *table_option* 值。（请参阅附录 E。）
  这个选项是默认启用的，但可以用--skip- create-options 选项禁用。

- --databases 或 -B（布尔）
  把 mysqldump 程序的所有参数都解释为数据库的名字，依次导出各指定数据库里的所有数据表。这个选项将使每个数据库的导出内容包含 CREATE DATABASE IF NOT EXISTS 和 USE 语句。
- --delayed-insert（布尔）
  编写 INSERT DELAYED 语句而不是 INSERT 语句。如果你要将 MyISAM 表的转储文件加载到其他数据库，并且想尽量减小在那个数据库上操作其他语句时产生的影响，那么采用这个选项可以达成目的。
- --delete-master-logs
  在生成转储输出文件后执行一条 FLUSH MASTER 语句，这将删除 MySQL 服务器上现有的二进制日志文件并开始一个新的。如果你对是否应该删除现有的二进制日志没有足够的把握，最好不要使用这个选项。这个选项将启用--master-date 选项。
- --disable-keys 或 -K（布尔）
  在 mysqldump 程序的输出内容里添加 ALTER TABLE...DISABLE KEYS 和 ALTER TABLE...ENABLE KEYS 语句，防止在处理 INSERT 语句时更新非唯一化索引。这将加快 MyISAM 数据表上的索引创建工作，因为只要加载了数据表，就会马上创建索引。
- --dump-date（布尔）
  添加一个注释，将转储日期指定到输出末尾。这个选项是从 MySQL 5.0.52/5.1.23 引入的。
- --dump-salve[=*n*]
  这个选项的效果类似于--master-data 选项，但它是用来转储复制机制中的从服务器的。它还将在转储输出文件里生成一条 CHANGE MASTER 语句以记录从服务器（请注意，不是从服务器本身）的二进制日志坐标。关于这个选项的参数的用法请参阅--master-data 选项条目里的有关描述。这个选项是从 MySQL 6.0.4 版开始引入的。
- --events 或-E（布尔）
  把事件也转储到输出文件里。这个选项是在 MySQL 5.1.8 中引入的。
- --extended-insert 或 -e（布尔）
  在导出文件里使用同时插入多个数据行的 INSERT 语句，这要比使用只插入单个数据行的 INSERT 语句更有效率。
- --first-salve 或 -x（布尔）
  这个选项已由--lock-all-tables 代替。
- --flush-logs 或 -F（布尔）
  在导出工作开始之前先清空 MySQL 服务器的日志文件。默认情况下，要清除每个数据库的日志，方便创建检查点。这样会更容易恢复操作，因为你知道在检查点时间之后创建的二进制日志文件是在备份给定数据库之后完成的。结合使用--lock-all-tables 或--master-data，只在所有数据表都锁定之后才清除日志。这个选项需要具备 RELOAD 权限。
- --flush-privileges（布尔）
  如果 mysql 数据库也包括在转储范围内，在转储完该数据库之后在输出文件里加上一条 FLUSH PRIVILEGES 语句。这个选项是从 MySQL 5.0.26/5.1.12 版开始引入的。
- --force 或-f（布尔）
  即使在转储过程中发生错误也要继续进行。

- ❑ --hex-blob（布尔）
  把 BINARY、VARBINARY 和 BLOB 数据列转储为十六进制常数。比如说，如果给出了这个选项，mysqldump 程序将把字符串值"MySQL"写成 0x4D7953514C。
- ❑ --ignore-table=*db_name.tbl_name*
  不对指定的数据表进行转储。如果需要跳过多个数据表，必须为每个数据表分别写出一个--ignore-table 选项。这个选项是从 MySQL 5.0.3 版开始引入的。
- ❑ --include-master-host-port（布尔）
  在使用--dump-slave 选项而生成的转储文件里，给 CHANGE MASTER 语句加上 MASTER_HOST 和 MASTER_PORT 选项以表明主服务器的主机名和端口号。这个选项是从 MySQL 6.0.4 版开始引入的。
- ❑ --insert-ignore（布尔）
  生成 INSERT IGNORE 语句而不是生成 INSERT 语句。这个选项是从 MySQL 5.0.6 版开始引入的。
- ❑ --lock-all-tables 或-x（布尔）
  用 FLUSH TABLES WITH READ LOCK 语句锁定将被转储的所有数据库里的所有数据表。这个选项将禁用--single-transaction 和--lock-tables 选项。
- ❑ --lock-tables 或 -l（布尔）
  在导出各有关数据表之前必须先使用 Lock TABLES...READ LOCAL 把它们都锁定住。这个选项对 MyISAM 表有益，因为一个 READ LOCAL 锁能让并发插入在转储过程中进行。对于 InnoDB 和 Falcon 表，--single-transaction 选项是首选。
- ❑ --log-error=*file_name*
  把警告消息和出错消息写到指定文件的末尾。这个选项是从 MySQL 5.0.42/5.1.18 版开始引入的。
- ❑ --master-data[=*value*]
  这个选项将使备份文件能够用于设置从服务器上。利用这个选项，mysqldump 将一个 SHOW MASTER STATUS 语句发送到服务器，以获取当前的二进制日志文件名和位置，并利用这个结果编写一个 CHANGE MASTER 语句发送到输出，其中包含同样的文件名和位置。这样做的效果就是，当你把转储文件加载到一个从服务器时，它将使从服务器与转储复制服务器同步，在转储时开始复制。只有在服务器启用了二进制日志功能时，这个选项才会有效果。
  默认情况下，CHANGE MASTER 语句是按不带注释的形式写成的。--master-data 采用了一个可选值来显式控制语句的注释。值为 1 时将生成一条不带注释的语句，值为 2 时将生成一条带注释的语句。
  --master-data 的执行需要你具备 RELOAD 权限。如果没有给定--single-transaction，这个选项自动启用--lock-all-tables。
- ❑ --no-autocommit（布尔）
  把对应于每一个数据表的全体 INSERT 语句生成为一个事务。与以自动提交模式执行这些 INSERT 语句相比，如此生成的导出文件将使你能够更有效率地完成数据加载工作。
- ❑ --no-create-db 或 -n（布尔）
  不在导出文件里写 CREATE DATABASE 语句。（当你使用了--databases 或--all-databases 选项时，mysqldump 程序就会自动地在导出文件里加上一些 CREATE DATABASE 语句。）
- ❑ --no-create-info 或 -t（布尔）
  不在导出文件里写出 CREATE TABLE 语句。如果你只想导出数据表里的数据，就需要使用这

个选项。

- --no-data 或 -d（布尔）

  不在导出文件里写出数据表的内容。如果只想导出 CREATE TABLE 语句，就需要使用这个选项。

- --no-tablespaces 或-y（布尔）

  不对表空间进行转储。这个选项是从 MySQL 5.1.14 版开始引入的。

- --opt

  加快数据表导出速度并生成一份能加快数据表导入操作速度的导出文件来。这个选项将自动启用以下选项（具体情况还要由你的 mysqldump 程序的版本来决定）：--add-drop-table、--add-locks、--create-options、--disable-keys、--extended-insert、--lock-tables 和--quick。这个选项是默认启用的，利用--skip-opt 可以禁用它。

- --order-by-primary（布尔）

  按照各数据表里的主键或者第一个唯一化索引（如果有的话）的顺序来转储其中的数据行。这将为每个数据表生成一份经过排序的转储输出，但代价是性能会降低。

- --quick 或 -q（布尔）

  默认情况下，mysqldump 程序先把一个数据表的内容全部读到内存里，然后再把它写出去。这个选项将使 mysqldump 在从服务器读到一个数据行之后立刻把它写到导出文件里去，这就大大减少了内存占用量。但如果你使用了这个选项，就不应该让 mysqldump 程序在执行中途被挂起来，那将使服务器进入等待状态，从而影响其他的客户（程序）的正常工作。

- --quote-names 或 -Q（布尔）

  把数据表和数据列的名字用反引号（`）字符括起来。如果这些名字里有 MySQL 保留字或者包含有特殊字符，这个选项就有用了。这个选项是默认启用的，使用--skip-quote-names 可以禁用它。

- --replace

  让 mysqldump 程序生成 REPLACE 语句而不是生成 INSERT 语句。

- --result-file=*file_name* 或 -r *file_name*

  把输出写到指定文件里去。这个选项是为基于 Windows 的系统准备的，它将防止一个换行符被自动转换为一个回车符加上一个换行符。

- --routines 或-R（布尔）

  把存储函数和存储过程也转储到输出文件里。这个选项是从 MySQL 5.0.13 版开始引入的。

- --set-charset（布尔）

  在输出文件里增加一条 SET NAMES *charset* 语句，其中 *charset* 的默认值是 utf8，该字符集可以用--default-character-set 选项来改变。这个选项是默认启用的，可以用--skip-set-charset 选项来禁用。

- --single-transaction（布尔）

  这个选项能够让 InnoDB 和 Falcon 数据表在备份过程中保持不变。这一做法的关键在于它是在同一个事务里来导出各有关数据表的。Mysqldump 使用 REPEATABLE READ 事务隔离层来生成一份稳定一致的转储文件，同时不会阻塞其他客户（对于非事务性表，转储过程可能有变化。）它不能与--lock-all-tables 选项一起使用。

- --skip-opt

  这个选项的使用效果和--opt 选项刚好相反。--opt 选项是默认启用的。

- `--tab=`*`dump_dir`* 或 `-T` *`dump_dir`*

  这个选项将使 mysqldump 程序为每个数据表生成两个导出文件并以 *dump_dir* 参数给定的子目录——这个子目录必须是已经存在的——作为这些文件的存放场所。对于每个名为 *tbl_name* 的数据表，mysqldump 程序将把它里面的数据保存到 *dump_dir/tbl_name*.txt 文件里去，把与之对应的 CREATE TABLE 语句保存到 *dump_dir/tbl_name*.sql 文件里去。此外，你还必须拥有相应的 FILE 权限才能使用这个选项。

  在默认情况下，数据文件中的每一行将以换行符结束，各数据列值之间以制表符分隔。这一格式可以通过后面 F.15.3 节里介绍的选项来加以改变。

  如果不了解--tab 选项的工作情况，就很容易把它的使用效果弄混淆。下面是使用--tab 选项时必须注意的两件事。

  - 有些文件将被写在服务器主机上，而另一些文件则将被写在客户主机上。*.txt 文件将被写到 MySQL 服务器主机的 *dump_dir* 子目录里，而*.sql 文件则将被写到客户主机的 *dump_dir* 子目录里。如果这两个主机不同，导出文件就将被分别创建在不同的机器里。因此，为避免让这些文件分散在不同的机器上，最好只在服务器主机上用--tab 选项来运行 mysqldump 程序。
  - *.txt 文件的属主将是你用来运行服务器程序的那个账户，而*.sql 文件的属主则是你用来运行 mysqldump 程序的那个账户。造成这一现象的原因是：*.txt 文件是由服务器亲自写出来的，其内容是数据表里的数据；而*.sql 文件则是由 mysqldump 程序写出来的，其内容是由服务器发送至 mysqldump 程序的 CREATE TABLE 语句。

- `--tables`

  覆盖--databases 选项，使以下参数被解释为表名。

- `--triggers`（布尔）

  把触发器也转储到输出文件里。触发器是默认转储的，你需要明确地给出--skip-trigger 选项才能把它们排除在外。这个选项是从 MySQL 5.0.11 版开始引入的。

- `--tz-utc`（布尔）

  让 mysqldump 程序在连接到服务器之后把地理时区设置为 UTC 并在输出文件里加上一条 SET TIME_ZONE='+00:00'语句。这么做的后果是在转储和重新加载数据的时候不对本地时区进行转换，因而可以确保 TIMEATAMP 值不会因为转储和重新加载操作发生在不同的时区而发生变化。这个选项是默认激活的，但你可以用--skip-tz-utc 选项禁用。这个选项是从 MySQL 5.0.15 版开始引入的。

- `--where =` *`where_expr`* 或`-w` *`where_expr`*

  只对满足 *where_expr* 给出的 WHERE 条件的数据行进行转储。应该把这个条件用引号括起来以防止你的命令解释器把它错误地解释为多个命令行参数。

- `--xml` 或 `-X`

生成 XML 格式的输出而不是一组 SQL 语句。

F.15.3　mysqldump程序的数据格式选项

如果你使用了--tab 或-T 选项，mysqldump 程序就会为每一个数据表分别生成一个数据文件。此时，你还可以使用以下几个额外的选项来进一步控制数据文件的格式。注意：你可能需要使用适当的引号字符把下面这些选项的值括起来。这些选项与 LOAD DATA 语句所使用的格式选项很相似，详细

情况请参见本书附录 D 中的 `LOAD DATA` 语句条目。

- `--fields-enclosed-by=`*`char`*
  用给定字符——通常是一个引号字符——来括住各数据列的值。这个选项的默认值是空字符，即不使用任何字符来括住各数据列的值。这个选项不能与`--fields-optionally-enclosed-by`选项同时使用。
- `--fields-escaped-by=`*`char`*
  把给定字符用作转义字符，即以它来开始对特殊字符进行转义的字符序列。这个选项的默认值是空，即不设定转义字符。
- `--fields-optionally-enclosed-by=`*`char`*
  用给定字符——通常是一个引号字符——来括住各非数值数据列的值。这个选项的默认值是空字符，即不使用任何字符来括住各非数值数据列的值。这个选项不能与`--fields-enclosed-by`选项同时使用。
- `--fields-terminated-by=`*`str`*
  在数据文件里，用给定字符串 *`str`* 来分隔各数据列的值。它可以是一个或者多个字符。默认的数据列值分隔符是制表符。
- `--lines-terminated-by=`*`str`*
  在数据文件里，用给定字符串 *`str`* 来分隔各输出行。它可以是一个或者多个字符。默认的输出行分隔符是换行符。

F.15.4　与mysqldump有关的变量

下面这些与 `mysqldump` 程序有关的变量可以按照 F.2.1 节中第 2 小节中的步骤进行设置。

- `max_allowed_packet`
  客户端与服务器通信时使用的缓冲区的最大长度。默认值为 24 MB，最大值为 1 GB。
- `net_buffer_length`
  客户端与服务器通信时使用的缓冲区的初始长度。这个缓冲区可以扩张到 `max_allowed_packet` 个字节长。默认值稍小于 1 MB。

F.16　mysqlhotcopy

`mysqlhotcopy` 工具脚本能够高效率地完成数据库和数据表的备份工作。它只能作用在 MyISAM 和 ARCHIVE 数据表上。`mysqlhotcopy` 是一个 Perl 脚本，它要求你的系统必须安装有 DBI 支持。（不要奇怪，因为它一开始是 Tim Bunce 编写的，他是 DBI 创建者之一。）`mysqlhotcopy` 只能在 Unix 和 Net Ware 上使用，它目前还不能用在 Windows 系统上。

`mysqlhotcopy` 脚本的工作流程是这样的：（1）连接本地主机上的服务器；（2）向服务器发出一系列数据表清空和锁定语以锁定各有关数据表；（3）把数据表文件复制到另一个地方作为备份。这将确保（1）把尚未写入磁盘的数据修改情况写到磁盘上去；（2）在备份数据表的过程中，服务器将不会对数据表做出新的修改。（也就是说，`mysqlhotcopy` 脚本实现了我们在 14.1 节中描述的协议——当你直接对数据表文件进行操作的时候，服务器将不去“打扰”你正在处理的数据表。）

这个脚本的使用方法有好几种。它的基本语法如下所示：

```
mysqlhotcopy [options] db_name[./regex/ ] [new_db_name | dir_name]
```

我们来看几个例子。下面这条命令将对数据库 *db_name* 进行备份，如果备份成功，在数据目录下就会增加一个名为 *db_name_copy* 的备份数据库：

```
% mysqlhotcopy [options] db_name
```

而下面这条命令将把数据库 *db_name* 复制到/tmp 子目录下一个名字仍为 *db_name* 的下级子目录里去：

```
% mysqlhotcopy [options] db_name /tmp
```

mysqlhotcopy 脚本的在线文档里有更多的例子；你可以用下面这条命令来查看它们：

```
% perldoc mysqlhotcopy
```

F.16.1 mysqlhotcopy支持的标准选项

```
--debug        --host         --port         --user
--help         --password     --socket
```

--host 选项（如果给出的话）只能用来指定本地主机的名字。在默认情况下，mysqlhotcopy 脚本将使用一个 Unix 套接字文件来连接本地主机上的服务器。如果你用--host 选项给出了该服务器的实名，mysqlhotcopy 脚本就将使用 TCP/IP 来建立连接；此时，你还可以用--port 选项来另行指定一个非默认的端口号。对于--password，密码值不是可取的。

F.16.2 mysqlhotcopy独有的选项

- --addtodest
  如果目标子目录已经存在，不要重新命名它，把备份文件添加到其中即可。
- --allowold
  如果目标子目录已经存在，在制作新备份之前先用“-old”后缀重新命名它。此后，如果备份工作失败，就要把那个子目录的名字再改回来；如果备份工作成功，则删除那个子目录——除非你还同时使用了--keepold 选项。
- --checkpoint=db_name.*tbl_name*
  把一个检查点记录写到指定的数据表里去，该数据表必须在事先用如下所示的语句创建好：

```
CREATE TABLE tbl_name
(
    time_stamp TIMESTAMP NOT NULL,
    src        VARCHAR(32),
    dest       VARCHAR(60),
    msg        VARCHAR(255)
);
```

  src 和 dest 是源数据库和目标数据库的名字，msg 是一条表明备份操作是否成功的消息。
- --chroot=*dir_name*
  如果 mysqld 程序运行在一个 chroot 环境里，应该使用这个选项来进行备份。*dir_name* 值是该 chroo 环境中的根目录的路径名。
- --dryrun 或 -n
  “不执行”模式。mysqlhotcopy 脚本将报告它都会采取哪些操作动作，但并不会真正地执行这些动作。你可以利用这个选项来验证 mysqlhotcopy 脚本的行为是不是与你想象的一样，这

为你学习它的使用方法提供了方便。

- --flushlog

  在各有关数据表全都被锁定之后、但在开始复制它们之前，先把缓存在内存里的日志信息写到磁盘上去。这相当于在复制操作开始之前对它们都进行一次检查点检查。

- --keepold

  如果目标子目录已经存在，在制作新备份之前先用-old 后缀重新命名它。这个选项隐含着--allowold 选项。

- --method=*copy_method*

  用来复制文件的方法。如果 *copy_method* 参数的值是 cp，则使用 cp 程序。现时期，还有一种实验性质的 scp 方法可供选用。在使用 scp 方法的时候，*copy_method* 参数值必须是你将使用的 scp 命令的完整内容，而且用来存放备份文件的目标子目录必须已经存在。因为通过网络来复制文件需要额外花费一些时间，所以 scp 方法会导致各有关数据表处于锁定状态的时间大为延长。为了避免这个问题，在本地备份，然后在 mysqlhotcopy 完成后把它复制到远程主机。

- --noindices

  不复制索引文件。如果今后需要使用如此备份出来的文件去恢复数据表，可以使用 myisamchk --recover（适用于 MyISAM 数据表）命令去重建索引。

- --quiet 或 -q

  只有在执行出错时才产生提示性输出信息。

- --record_log_pos=*db_name.tbl_name*

  在开始复制数据表之前， 先发出 SHOW MASTER STATUS 和 SHOW SLAVE STATUS 语句并把它们的执行结果记录到指定的数据表里去，该数据表必须在事先用如下所示的语句创建好：

```
CREATE TABLE tbl_name
 (
    host            VARCHAR(60) NOT NULL,
    time_stamp      TIMESTAMP NOT NULL,
    log_file        VARCHAR(32) NULL,
    log_pos         INT NULL,
    master_host     VARCHAR(60) NULL,
    master_log_file VARCHAR(32) NULL,
    master_log_pos  INT NULL,
    PRIMARY KEY (host)
);
```

  SHOW MASTER STATUS 语句的执行结果将被记录在 log_file 和 log_pos 数据列里，它们提供了主服务器上的二进制日志的同步信息。如果备份主机是复制机制中的主服务器，用它的备份文件初始化出来的从服务器将从这些同步信息所指定的位置开始去进行下一次复制同步工作。SHOW SLAVE STATUS 语句的执行结果将被记录在 master_host、master_log_file 和 master_log_pos 数据列里。如果备份主机是复制机制中的从服务器而你打算用备份文件去对同一主服务器的另一个从服务器进行初始化，就需要用到这些信息了。

- --regexp=*pattern*

  只对名字与给定的正则表达式相匹配的那些数据库进行复制。如果使用了这个选项，mysqlhot-copy 命令行上的最后一个参数就必须是你将用来存放备份数据库的那个子目录的名字。

- --resetmaster

在所有数据表被锁定之后、但在开始复制它们之前，先发出一条 RESET MASTER 语句对二进制日志进行重置。

- `--resetslave`
  在所有数据表被锁定之后、但在开始复制它们之前，先发出一条 RESET SLAVE 语句对 master.info 文件中的信息进行重置。
- `--suffix=`*str*
  如果你打算把备份数据库与原始数据库保存在同一个子目录里，就需要使用这个选项给备份数据库的名字加上一个后缀以区分这二者。新数据库子目录名字的前半截与原始数据库子目录的名字相同，后半截则是你用这个选项给出的字符串。
- `--tmpdir=`*dir_name*
  用来存放临时文件的子目录的路径名。这个选项的默认值是环境变量 TMPDIR 所指定的子目录，如果该变量没有定义，则将使用/tmp 作为临时子目录。

F.17 mysqlimport

mysqlimport 程序是一个大批量数据的加载工具，它可以把文本文件的内容读到现有的数据表里去。它的功能相当于 SQL 语句 LOAD DATA 的一个命令行接口，是一种能够非常高效地把数据行加载到数据表里去工具。

```
mysqlimport [options] db_name file_name...
```

mysqlimport 程序将把数据加载到 *db_name* 参数给出的数据库所包含的数据表里，这个数据表将由文件名参数 *file_name* 确定。对于每一个文件名，从第一个句点开始的后半部分将被截去，剩余部分将将被视为一个数据表名，而数据就将被加载到这个数据表里去。比如说，mysqlimport 程序将把 president.txt 文件的内容加载到 president 数据表里去。

mysqlimport 程序只能读取数据文件。它不能读取 mysqldump 程序生成的 SQL 格式的导出文件，SQL 格式的导出文件要用 mysql 程序来读取。

F.17.1 mysqlimport支持的标准选项

```
--character-sets-dir      --help        --shared-memory-base-name
--compress                --host        --silent
--debug                   --password    --socket
--debug-check             --pipe        --user
--debug-info              --port        --verbose
--default-character-set   --protocol    --version
```

--debug-info 和--debug-check 选项分别始见于 MySQL 5.1.14 和 MySQL 5.1.21 版本。

mysqlimport 程序还支持各种标准的 SSL 选项。

F.17.2 mysqlimport独有的选项

下面介绍的各个选项都是用来控制 mysqlimport 程序的操作行为的。在随后的 F.13.3 节里，我们将对那些用来表明输入文件格式的选项做集中的介绍。

- `--columns=`*col_list*
  col_list 是由数据表里的数据列名构成的清单，数据文件里的数据将依次被加载到这些数据列里去，而没有在 *col_list* 里出现的数据列将被设置为它们的默认值。在 *col_list* 里，数

据列名之间要用逗号隔开。

- --delete 或 -d（布尔）

  在把数据加载到每一个数据表里之前，先彻底清除该数据表里的现有内容。
- --force 或 -f（布尔）

  不管执行是否出错，都继续加载数据。
- --ignore 或 -i

  如果输入行里的某个值会导致数据表的唯一化索引出现重复的键值，则保留现有的数据行而丢弃来自数据文件的输入行。--ignore 与--replace 选项是互相排斥的，不能同时使用。
- --ignore-lines=*n*

  忽略（不加载）数据文件的前 *n* 个输入行。这个选项的用途之一是跳过数据文件开头部分的数据列标题行。
- --local 或 -L（布尔）

  在默认情况下，mysqlimport 程序会让服务器去读取数据文件，这意味着数据文件必须存放在服务器主机上，并且你还必须具备相应的 FILE 权限。如果你使用了--local 选项，mysqlimport 程序就将亲自读取数据文件并把读到的数据发送给服务器。后一种方式的数据加载速度比较慢，但它的优势是允许你（1）在服务器主机以外的其他机器上运行 mysqlimport 程序；（2）即便是在服务器主机上，也不要求你必须具备相应的 FILE 权限。

  如果 MySQL 服务器被配置成不允许使用 LOAD DATA LOCAL 语句的情况，这个选项将没有效果。
- --lock-tables 或 -l（布尔）

  在把数据加载到数据表里之前，先锁定之。
- --low-priority（布尔）

  使用优先级调整符往数据表里加载数据。
- --replace 或 -r（布尔）

  如果输入行里的某个值会导致数据表的唯一化索引出现重复的键值，则用来自数据文件的输入行替换掉现有的数据行。--ignore 与--replace 选项是互相排斥的，不能同时使用。
- --use-threads =*n*

  同时使用 *n* 个线程来加载文件。这个选项是从 MySQL 5.1.7 版开始引入的。

F.17.3 **mysqlimport**程序的数据格式选项

在默认情况下，mysqlimport 程序将假设数据文件中的每一行将以换行符结束，各数据列值之间以制表符分隔。但这可以通过以下选项加以改变。你可能需要使用适当的引号字符把下面这些选项的值括起来。这些选项与 LOAD DATA 语句所使用的格式选项很相似，详细情况请参见本书附录 D 中的 LOAD DATA 语句条目。

- --fields-enclosed-by=*char*

  表明各数据列的值是用给定字符——通常是一个引号字符——来括住的。这个选项的默认值是空字符，即没有使用任何字符来括住各数据列的值。这个选项不能与--fields-optionally-enclosed-by 选项同时使用。
- --fields-escaped-by=*char*

  表明给定字符 *char* 是转义字符，它对特殊字符进行转义。这个选项的默认值是空，即没有设定转义字符。

- --fields-optionally-enclosed-by=*char*
  表明数据列的值是用给定字符——它通常是一个引号字符——来括住的。这个选项不能与--fields-enclosed-by 选项同时使用。
- --fields-terminated-by=*str*
  在数据文件里，各数据列的值是用给定字符串 *str* 来分隔的。它可以是一个或者多个字符。默认的数据列值分隔符是制表符。
- --lines-terminated-by=*str*
  在数据文件里，各输入行是用给定字符串 *str* 来分隔的。它可以是一个或者多个字符。默认的输入行分隔符是换行符。

F.18 mysqlshow

你可以用 mysqlshow 程序查知系统中有哪些数据库、每个数据库里容纳着哪些个数据表、每个数据表里又有哪些数据列或索引。它相当于 SQL 语句 SHOW 的一个命令行接口。

```
mysqlshow [options] [db_name [tbl_name [col_name]]]
```

如果你没有在 mysqlshow 命令行上给出数据库名，mysqlshow 将列出服务器主机上的所有数据库。如果给出了数据库名但没有给出数据表名，mysqlshow 将列出该数据库里的所有数据表。如果给出了数据库名和数据表名但没有给出数据列名，它将列出该数据表里所有数据列。如果给出了所有的名字，mysqlshow 将把给定数据列的描述信息显示出来。

如果最后一个参数中包含有通配符（“%”或“-”），mysqlshow 程序将把它们当做 LIKE 操作符所使用 SQL 通配符“%”和“_”来对待，输出将仅限于与该通配符相匹配的数据库、数据表或数据列。如果最后一个参数包含“*”或“?”通配符，它们将被当做“%”和“-”。

F.18.1 mysqlshow支持的标准选项

```
--character-sets-dir      --help          --shared-memory-base-name
--compress                --host          --socket
--debug                   --password      --user
--debug-check             --pipe          --verbose
--debug-info              --port          --version
--default-character-set   --protocol
```

--debug-info 和--debug-check 选项是从 MySQL 3.23.21 版本开始增加的。

如果给出了--verbose 选项，mysqlshow 程序将额外增加一些输出列（关于给定数据库里的各数据表、关于给定数据表里的各数据行，等等）的信息。这个选项可以多次给出。

mysqlshow 程序还支持标准的 SSL 选项。

F.18.2 mysqlshow程序独有的选项

- --count（布尔）
  对每个数据表里的数据行进行计数并把它显示在输出报告里。这对某些存储引擎来说是一个比较慢的操作。只有当你指定了一个数据名时这个选项才有意义。
- --status 或 -i（布尔）
  这个选项显示的信息与你使用 SHOW TABLE STATUS 语句查到的信息是一样的。
- --keys 或 -k（布尔）
  除关于数据列的描述信息外，这个选项还能让你看到关于数据表索引的描述性信息。这个选项

只有在你给出了数据表名参数的时候才有意义。

F.19　perror

perror 程序的用途是查看某给定出错代码所对应的出错消息：

```
perror [options] [err_code] ...
```

你可以用它来确定 MySQL 程序所返回的出错代码的的含义。

```
% perror 142
MySQL error:  142 = Unknown character set used
```

perror程序支持的标准选项

```
--help        --silent      --verbose     --version
```

--silent 选项将使 perror 程序只显示出错消息，不显示出错代码。默认设置是--verbose 选项，也就是出错代码和出错消息两者都要显示。

--info 和-I 选项是--help 选项的同义词。

附录 G API 指南

本附录对 MySQL C 客户程序开发库所提供的 C 语言应用程序设计接口进行了介绍。这个 API 由一组用来与 MySQL 服务器进行通信和访问数据库的函数以及一组供这些函数使用的数据类型构成。这个客户程序开发库里的函数可以划分为以下几大类。

- 对客户库进行初始化和收尾处理的例程。
- 连接管理类例程，用来建立和断开与服务器的连接。
- 出错报告类例程，用来获得出错代码和出错信息。
- 查询构造与执行类例程，用来构造 SQL 语句并把它们发送给服务器。
- 结果集处理类例程，用来处理从服务器返回的查询结果数据。
- 信息收集类例程，用来收集关于客户程序、服务器、协议版本、当前连接的信息。
- 事务控制例程。
- 多结果集处理例程。
- 服务器端 SQL 语句预处理例程。
- 用来调控服务器操作的系统管理例程。
- 用来编写线程化客户程序的线程例程。
- 调试信息生成例程。

如无特别说明，本附录所介绍的数据类型和函数至少从 MySQL 5.0.0 版本开始就已经存在于 MySQL C 客户程序开发库里了。自那时起发生的改变会在本书提醒你注意。

本附录示例收录的都是一些代码片段。在本书的第 7 章大家可以看到一些完整的客户程序以及编写它们的指导意见。

G.1 编译和链接

在源代码层次上， C 客户程序开发库的调用接口是在头文件里定义的。在自行编写 MySQL 客户程序时，你的源代码文件里必须有下面 3 条指令：

```
#include <my_global.h>
#include <my_sys.h>
#include <mysql.h>
```

为了让编译器能够找到这个文件，你可能需要给出一个`-Ipath_name` 选项，其中的 *path_name* 是存放各种 MySQL 头文件的子目录的路径名。比如说，如果你已经把 MySQL 头文件安装在了/usr/include/mysql 或者/usr/local/mysql/include 子目录里，那么，当你想对源代码文件 my_func.c 进行编

译时，就需要使用下面这样的命令：

```
% gcc -I/usr/include/mysql -c my_func.c
% gcc -I/usr/local/mysql/include -c my_func.c
```

如果还需要用到其他 MySQL 头文件，可以在 mysql.h 文件所在的子目录里找到它们。比如说，头文件 mysql_com.h 里包含着解释结果集元数据时需要用到的常数和宏，头文件 errmsg.h 和 mysqld_error.h 里包含着解释出错代码时需要用到的常数。（注意：虽然你可能需要去看看 mysql_com.h 文件里都有哪些内容，但你并不需要明确地把它导入到你的源代码文件里，这件事可以让 mysql_com.h 代劳。只需导入 mysql_com.h 文件，你的程序就能自由地访问 mysql_com.h 文件所定义的东西了。）

MySQL 程序可以大致划分为两种，一种是能够使用通用的客户/服务器协议去连接一个独立式 MySQL 服务器并与之通信的客户程序，另一种则是把一个嵌入式服务器直接链接在二进制可执行代码里的一体化应用程序。这两种程序在编程阶段没有什么差异，只要你在使用 C API 的 mysql_library_init()初始化例程和 mysql_library_end()收尾处理例程时没犯错误，你编写出来的程序就应该能够同时支持独立式和嵌入式服务器。（请参阅 G.3.1 节。）你选用哪个库和你的程序代码相链接以生成可执行图像决定了程序将支持哪种服务器。

- 如果你选用 libmysqlclient 库和你的程序代码相链接，你将得到一个客户程序。在链接命令行上给出-lmysqllcient 选项就可以把这个库链接到你的程序里。你可能还需要使用 -L*path_name* 选项告诉链接器应该去什么地方寻找这个库，其中 *path_name* 是这个库的安装子目录的路径名。如下所示：

```
% gcc -o myprog my_main.o my_func.o -L/usr/local/mysql/lib
    -lmysqlclient
```

- 如果你选用 libmysqld 库和你的程序代码相链接，你将得到一个内嵌服务器的一体化应用程序。在链接命令行上给出-lmysqld 选项就可以把这个库链接到你的程序里，如下所示：

```
% gcc -o myprog my_main.o my_func.o -L/usr/local/mysql/lib -lmysqld
```

如果链接命令没有执行成功并给出了“unresolved symbol”（无法对符号作出解析）出错信息，就说明你还得再列举几个函数库——比如数学函数库（-lm）和 zlib 函数库（-lz-lgz）——供链接器去搜索。

mysql-config 程序是个非常有用的工具，它可以帮我们迅速确定最适用的头文件子目录和链接库选项。下面几个例子演示了如何使用这个小工具去查出适用于你系统的有关选项或设置。

- 查编译选项：

```
% mysql_config --include
-I'/usr/local/mysql/include/mysql'
```

- 查用来制作客户程序的链接选项：

```
% mysql_config --libs
-L'/usr/local/mysql/lib/mysql' -lmysqlclient -lz -lcrypt -lnsl -lm
```

- 查用来链接内嵌式服务器的链接选项：

```
% mysql_config --libmysqld-libs
-L'/usr/local/mysql/lib/mysql' -lmysqld -lz -lcrypt -lnsl -lm
```

上面几个例子里的输出只是演示性的，和你自己系统上的可能会有所不同。

G.2 C API 数据类型

在与服务器的会话过程中，你需要用到和处理各种各样的信息，MySQL 客户程序开发库所提供的数据类型就是为了表示这些信息而准备的。这些数据类型分别对应着会话连接本身、查询结果、结果集里的数据行、元数据（即同一结果集里的数据列的描述性信息），等等。在下面的讨论里，术语“数据列”和“字段”的所代表的含义将是相同的。

G.2.1 标量数据类型

MySQL 用以下几种标量数据类型来表示非常大的整数、布尔值、字段或数据行偏移量。

- my_bool

  一种布尔类型，用来表示 mysql_change_user()、mysql_thread_init()等函数的返回值。
- my_ulonglong

  一种长整数类型，用来表示 mysql_affected_rows()、mysql_num_rows()、mysql_insert_id()等函数所返回的数据行计数值或其他有可能很巨大的数字。如果想把一个 my_ulonglong 值打印出来，应该把它转换为 unsigned long 类型并使用%lu 格式符，如下所示：

  ```
  printf ("Row count = %lu\n", (unsigned long) mysql_affected_rows (conn));
  ```

  在某些系统上，如果不这样做，就无法把这种值正确地打印出来，这是因为没有标准对如何使用 printf()来打印 long long 值作出规定，不同的系统有着不同的做法。不过，要是想打印的值已经超过了 unsigned long 类型所能表示的最大数值（$2^{32}-1$），那么%lu 格式符也将无法工作。

  printf()函数的不同版本在超大数值的打印方面有着不同的实现方法，请大家自行查阅自己的 printf()文档以了解其具体做法。比如说，某些系统允许你使用%llu 格式符来打印超大数值。
- MYSQL_FIELD_OFFSET

  这种数据类型是为 mysql_fields_seek()和 mysql_fields_tell()函数而准备的，用来表示当前结果集的各 MYSQL_FIELD 结构的偏移量。
- MYSQL_ROW_OFFSET

  这种数据类型是为 mysql_rows_seek()和 mysql_rows_tell()函数而准备的，用来表示当前结果集里的各数据行的偏移量。

G.2.2 非标量数据类型

MySQL 用以下几种非标量数据类型来表示结构或数组。需要特别注意的是，MYSQL、MYSQL_RES 或 MYSQL_STMT 结构类型的任何一个实例都必须被视为一个“黑箱”。也就是说，你只能引用该结构本身而不能引用该结构里的元素。MYSQL_ROW、MYSQL_FIELD、 MYSQL_BIND 和 MYSQL_TIME 类型没有这样的限制，你可以随意访问这两种结构里的各个元素以获取查询结果集里的数据和元数据。MYSQL_BIND 和 MYSQL_TIME 结构也可以用于将数据传递到服务器和从服务器获取结果。

- MYSQL

  这是 MySQL 客户程序开发库所提供的基本类型之一，用来表示连接句柄。每个连接句柄包含着客户程序与 MySQL 服务器之间的某一条连接的状态信息。要想与 MySQL 服务器进行会话，

就得先用 mysql_init()函数去初始化一个 MYSQL 结构，然后再把它传递给 mysql_real_connect()函数。在成功地建立起与 MySQL 服务器的连接之后，发出查询命令、生成结果集、获取出错信息等工作都必须通过该连接的连接句柄来进行。当你准备结束这次会话时，还得把连接句柄传递给 mysql_close()函数以关闭之；此后，这个句柄就不再有效了。

- MYSQL_FIELD

  客户程序开发库使用 MYSQL_FIELD 结构来存放结果集里的数据列的元数据，每个数据列都有一个与之对应的 MYSQL_FIELD 结构。结果集里的 MYSQL_FIELD 结构的个数可以调用 mysql_num_fields()函数去查知。你可以使用 mysql_fetch_field()函数去依次访问各个字段的 MYSQL_FIELD 结构，或者利用 mysql_field_tell()和 mysql_field_seek()函数快速前进/后退到某个 MYSQL_FIELD 结构上去。

  了解 MYSQL_FIELD 结构对表示或者解释数据行的内容有着重要的意义。下面是它的定义情况：

```
typedef struct st_mysql_field {
  char *name;
  char *org_name;
  char *table;
  char *org_table;
  char *db;
  char *catalog;
  char *def;
  unsigned long length;
  unsigned long max_length;
  unsigned int name_length;
  unsigned int org_name_length;
  unsigned int table_length;
  unsigned int org_table_length;
  unsigned int db_length;
  unsigned int catalog_length;
  unsigned int def_length;
  unsigned int flags;
  unsigned int decimals;
  unsigned int charsetnr;
  enum enum_field_types type;
} MYSQL_FIELD;
```

MYSQL_FIELD 结构的各个成员的含义如下所示。

- char *name

  结果集中的数据列的名字，它是一个以空结尾的字符串。如果该数据列是某个表达式的计算结果，它的 name 值将是该表达式的字符串形式。如果该数据列或表达式是用一个别名给出的，它的 name 值就将是那个别名。比如说，与下面这个查询命令相对应的各个 name 值将是"mycol"、"4*(mycol+1)"、"mc"和"myexpr"：

```
SELECT mycol, 4*(mycol+1), mycol AS mc, 4*(mycol+1) AS myexpr ...
```

- char *org_name

  这个成员与 name 相似，但数据列别名将被忽略。换句话说，org_name 里是数据列真正的名字。如果该数据列是某个表达式的计算结果，它的 org_name 值将是一个空字符串。

- char *table

  该数据列所在的数据表的名字，它是一个以空结尾的字符串。如果该数据列是视图，那么 table

就是视图名。如果该数据表或视图是用一个别名而给出的，它的 table 值就将是那个别名。如果该数据列是某个表达式的计算结果，它的 table 值将是一个空字符串。比如说，如果你发出了下面这个查询，那么第一个数据列的 table 值就将是 mytbl，而第二个数据列的 table 值将是一个空字符串：

```
SELECT mycol, mycol+0 FROM mytbl ...
```

- char *org_table
  这个成员与 table 相似，但数据表别名将被忽略。换句话说，org_table 里是数据表真正的名字。如果数据列取自一个视图，那么 table 就是表名。如果该数据列是某个表达式的计算结果，它的 org_table 值将是一个空字符串。
- char *db
  包含该数据列的数据表所在的数据库的名字，它是一个以空结尾的字符串。如果该数据列是某个表达式的计算结果，它的 db 值将是一个空字符串。
- char *catalog
  所属类别的名字。就现阶段而言，这个值总是"def"。
- char *def
  该数据列的默认值，它是一个以空结尾的字符串。MYSQL_FIELD 结构的这个成员只有在结果集是通过 mysql_list_fields()函数（已弃用）调用而获得的的时候才会被设置；它在其他情况下的取值将是 NULL。
  表的默认列值可以通过执行 DESCRIBE tbl_name 或 SHOW COLUMNS FROM *tbl_name* 语句并检查结果集来获取。
- unsigned long.length
  该数据列的长度，也就是你在创建数据表时在 CREATE TABLE 语句里为该数据列设定的长度。如果该数据列是某个表达式的计算结果，它的 length 值将由该表达式中的各个元素来确定。
- unsigned long max_length
  该数据列在结果集里的最长的值的长度。比如说，如果某字符串数据列在结果集里的值分别是"Bill"、"Jack"和"Belvidere"，它的 max_length 值就将是 9。
  结果集以字符串返回，所以这个长度涉及结果中值的最长字符串表示，即使是对于非字符串数据列。
  注意，因为 max_length 值只有在所有数据行都被看过之后才能确定，所以它只对你使用 mysql_store_result()函数而创建出来的结果集才有实际意义。对于使用 mysql_store_result()函数创建出来的结果集，max_length 值将是 0。
- unsigned int name_length, org_name_length, table_length, org_table_length,db_length, catalog_length, def_length
  分别是 name、org_name、table、org_table、db、catalog 和 def 成员的长度。
- unsigned int flags
  flags 成员记录着该数据列的属性，每种属性对应着 flags 值里的一个比特位，你可以用表 G-1 列出的位掩码常数来测试它们。比如说，如果你想知道某数据列的值是不是 UNSIGNED，就要按如下所示测试 flags 值：

```
if (field->flags & UNSIGNED_FLAG)
  printf ("%s values are UNSIGNED\n", field->name);
```

表G-1　**MYSQL_FIELD flags**成员值

flags值	含　义
AUTO_INCREMENT_FLAG	该数据列具备AUTO_INCREMENT属性
BINARY_FLAG	该数据列具备BINARY属性
MULTIPLE_KEY_FLAG	该数据列是某个非唯一化索引的组成部分
NOT_NULL_FLAG	该数据列不允许包含NULL值
NO_DEFAULT_VALUE_FLAG	该数据列定义没有DEFAULT子句
NUM_FLAG	该数据列类型为数值
PRI_KEY_FLAG	该数据列是某个PRIMARY KEY的组成部分
UNIQUE_KEY_FLAG	该数据列是某个UNIQUE索引的组成部分
UNSIGNED_FLAG	该数据列具备UNSIGNED属性
ZEROFILL_FLAG	该数据列具备ZEROFILL属性

如果数据列的类型为 MYSQL_TYPE_DECIMAL、MYSQL_TYPE_TINY、MYSQL_TYPE_SHORT、MYSQL_TYPE_LONG、MYSQL_TYPE_FLOAT、MYSQL_TYPE_DOUBLE、MYSQL_TYPE_NULL、MYSQL_TYPE_TIMESTAMP、MYSQL_TYPE_LONGLONG、MYSQL_TYPE_INT24 或 MYSQL_TYPE_YEAR.，那么 NUM_FLAG 为 true。

如果数据列定义中没有 DEFAULT 子句，那么 NO_DEFAULT_VALUE_FLAG 为 true，除非数据列允许为 NULL 或者具有 AUTO_INCREMENT 属性。这种数据列分别隐式默认为 NULL 和下一个序列值。NO_DEFAULT_VALUE_FLAG 是从 MySQL 5.0.2 开始引入的。

此外，还有几个用来表明数据列类型而非数据列属性的位掩码常数，但它们现在已经被逐渐淘汰了，你应该用 field->type 来确定数据列的类型。表 G-2 列出了这些已逐渐被淘汰的常数。

表G-2　已逐渐被淘汰的**MYSQL_FIELD flags**位掩码常数及其含义

flags值	含　义
BLOB_FLAG	该数据列包含BLOB或TEXT值
ENUM_FLAG	该数据列包含ENUM值
SET_FLAG	该数据列包含SET值
TIMESTAMP_FLAG	该数据列包含TIMESTAMP值

- unsigned int decimals

  数值数据列的小数部分的位数，非数值数据列的这个值是 0。比如说，DECIMAL(8, 3)的 decimal 值是 3，而一个 BOLB 数据列的 decimal 值是 0。
- unsigned int charsetnr

  字符集/字符序列数目。如果需要知道一个字符串数据列包含的是二进制还是非二进制字符数据，可通过 charsetnr 来确定，对二进制字符串来说，charsetnr 为 63。
- enum enum_field_types_type

  该数据列的类型。如果该数据列是某个表达式的计算结果，它的 type 值将根据该表达式各个元素的类型来确定。比如说，如果 mycol 是一个 VARCHAR(20)数据列，它的 type 值就将是 FIELD_TYPE_VAR_STRING，而 LENGTH(mycol)的 type 值却是 MYSQL_TYPE_LONGLONG。type 成员的可取值见表 G-3，它们都是在 mysql_com.h 文件里定义的。

表G-3 **MYSQL_FIELD type**成员的可取值

type值	含 义
MYSQL_TYPE_TINY	TINYINT
MYSQL_TYPE_SHORT	SMALLINT
MYSQL_TYPE_INT24	MEDIUMINT
MYSQL_TYPE_LONG	INT
MYSQL_TYPE_LONGLONG	BIGINT
MYSQL_TYPE_DECIMAL	DECIMAL, NUMERIC
MYSQL_TYPE_NEWDECIMAL	DECIMAL, NUMERIC
MYSQL_TYPE_DOUBLE	DOUBLE, REAL
MYSQL_TYPE_FLOAT	FLOAT
MYSQL_TYPE_STRING	CHAR
MYSQL_TYPE_VAR_STRING	VARCHAR
MYSQL_TYPE_BLOB	BLOB, TEXT
MYSQL_TYPE_ENUM	ENUM
MYSQL_TYPE_SET	SET
MYSQL_TYPE_DATE	DATE
MYSQL_TYPE_DATETIME	DATETIME
MYSQL_TYPE_TIME	TIME
MYSQL_TYPE_TIMESTAMP	TIMESTAMP
MYSQL_TYPE_YEAR	YEAR
MYSQL_TYPE_GEOMETRY	空间类型
MYSQL_TYPE_BIT	BIT
MYSQL_TYPE_NULL	NULL

从 MySQL 5.0.3 起，对于 DECIMAL 或 NUMERIC 值，返回值为 MYSQL_TYPE_NEWDECIMAL 类型，而此前返回的是 MYSQL_TYPE_DECIMAL。

- MYSQL_RES

  SELECT 或 SHOW 查询将以结果集的形式把数据返回给客户程序，这个 MYSQL_RES 结构就是用来表示各种结果集的。数据库查询命令所返回的数据行信息都包含在相应的 MYSQL_RES 结构里。在获得结果集之后，你可以调用相应的 API 函数来提取该结果集的数据部分（即该结果集中各数据行的数据值）和元数据部分（关于结果集的描述性信息，比如有多少个数据列、它们的类型、它们的长度，等等）。

- MYSQL_ROW

  MYSQL_ROW 类型包含着结果集里的一个数据行的值，这些值被表示为一个字符串数组。所有的值都将被返回为字符串形式（数值也不例外）；但如果数据行中的某个值是 NULL，那它在 MYSQL_ROW 结构里就将被表示为一个 C 语言中的 NULL 指针。

  你可以用 mysql_num_fields()函数来查知每个数据行里都有多少个值(即有多少个数据列)。第 *i* 个数据列的值可以通过 row[*i*]提取出来；下标 *i* 的取值范围是 0 到 mysql_num_fields(res_set)-1，res_set 是一个指向相应的 MYSQL_RES 结果集的指针。

  注意，MYSQL_ROW 类型已经是一个指针了，所以你应该像下面这样去定义一个 row 变量：

  ```
  MYSQL_ROW  row;        /* correct */
  ```

  你不能像下面这样做：

  ```
  MYSQL_ROW  *row;       /* incorrect */
  ```

  MYSQL_ROW 数组里的值都是以空结尾的，所以可以把非二进制的数据值当做以空结尾的字符

串来处理。但如果数据值本身就包含有二进制数据，其中就很可能会出现空，你必须把这种数据值当做计数字符串来对待。mysql_fetch_lengths()函数能够让你得到一个数组，这个数组的各个元素是结果集里的某一个数据行的各个值（即各个数据列值）的长度，它的使用方法如下所示：

```
unsigned long *length;
length = mysql_fetch_lengths (res_set);
```

数据行中第 *i* 个数据列值的长度可以通过 length[*i*]提取出来。如果某个数据列值是 NULL，它的长度就将是 0。

- MYSQL_STMT

  一个预处理语句处理程序。这种处理程序是通过调用 mysql_stmt_init()函数而创建的，这个函数将返回一个指向新处理程序的指针，你可以在随后的代码里通过这个指针对语句进行预处理、执行，等等。在用完某个指针之后，把它传递到 mysql_stmt_close()函数以释放之，释放后的指针将不能再使用了。

- MYSQL_BIND

  这种结构是和预处理语句配合使用的，它们既可以用于输入，也可以用于输出。

  在用于输入时，包含在 MYSQL_BIND 结构里的数据将被传递给服务器并绑定到将要执行的预处理语句的参数上。其基本流程是：先创建一个结构数组，然后调用 mysql_stmt_bind_param()函数把该数组里的各个元素绑定到那条将要执行的语句，再调用 mysql_stmt_execute()函数执行那条语句。那条语句有多少个参数，结构数组就必须包含多少个元素。

  输入字符串默认使用 character_set_client 系统变量指定的字符集来表示。如果这个字符集和相关数据列实际使用的字符集不一样，输入数据将被转换为以后者表示，这种转换发生在服务器端。

  在用于输出时，当一条会生成一个结果集的预处理语句执行完毕后，你需要使用 MYSQL_BIND 结构从那个结果集里把数据值提取出来。其基本流程是：先创建一个结构数组，然后调用 mysql_stmt_bind_result()函数把该数组里的各个元素绑定到那条执行完毕的语句，再调用 mysql_stmt_fetch()函数取回其结果集里的数据行。那个结果集有多少个数据列，你为它而创建的 MYSQL_BIND 结构数组就必须包含多少个元素。

输出字符串使用 character_set_result 系统变量指定的字符集来表示。

  MYSQL_BIND 结构包含许多个成员，但它们当中只有一部分允许从外部访问。如下所示的定义列出了 MYSQL_BIND 结构里允许从外部访问的成员：

```
typedef struct st_mysql_bind
{
  unsigned long         *length;
  my_bool               *is_null;
  void                  *buffer;
  my_bool               *error;
  unsigned long         buffer_length;
  enum enum_field_types buffer_type;
  my_bool               is_unsigned;
  ...
} MYSQL_BIND;
```

每个 MYSQL_BIND 结构将被绑定到相关预处理语句的一个参数上。下面对 MYSQL_BIND 结构里的各个成员在该结构用于输入和输出时的用途进行了描述；在讨论中，“真”值表明那是一个非零值，

“假”值表明那是一个零值。

- enum enum_field_types buffer_type
  绑定到给定参数的 C 语言变量的数据类型。这个成员必须被设置为一个 MYSQL_TYPE_XXX 值。
  在用于输入时，buffer_type 是用来向服务器传递参数值的变量的类型。
  在用于输出时，buffer_type 是用来从服务器接收结果值的变量的类型。
  表 G-4 和表 G-5 分别列出了当 MYSQL_BIND 结构用于输入和输出时与各种 C 变量数据类型相对应的 buffer_type 值。在输入和输出这两个方向里，如果 C 变量的类型和服务器端数据值的 SQL 类型不兼容，就需要进行类型转换。如果 C 变量的类型和 SQL 数据值的类型直接兼容，就不需要进行转换，而这意味着更好的性能。
- void *buffer
  这个指针指向用来发送或接收数据值的变量。
  在用于输入时，这个指针指向用来向服务器发送参数值的变量。
  在用于输出时，这个指针指向用来从服务器接收结果值的变量。
  buffer 永远是给定存储变量的地址。对于数值类型，buffer 指向一个标量变量。对于字符串类型，buffer 指向一个 char 缓冲区。对于日期/时间类型，它指向一个 MYSQL_TIME 结构。给定变量的类型由 buffer_type 值表明。如果变量是 unsigned 型的，is_unsigned 值应该设置为“真”。
- unsigned long buffer_length
  buffer 指向的缓冲区实际长度，以字节计算，不区分输入或输出用途。这适用于字符串类型（二进制或非二进制的均可，长度也允许变化），还可以用来输出 BIT 值。对于其他数据类型，这个缓冲区的长度取决于 buffer_type 值。
- unsigned long *length
  这个指针指向的变量保存着被传输的数据值里的字节个数。类似于 buffer_length，这个成员只在处理字符串类型的数据以及输出 BIT 值的时候才需要设置。对于数值和日期/时间类型，这个长度取决于具体的数据类型。
  在用于输入时，应该把这个指针所指向的变量设置为将被发送到服务器的字节数。
  在用于输出时，这个指针所指向的变量将由 mysql_stmt_fetch() 函数设置，而这个函数的返回值决定了应该如何解释变量值。如果 mysql_stmt_fetch() 函数返回 0（调用成功），*length 是被返回的数据值的实际长度。如果 mysql_stmt_fetch() 函数返回 MYSQL_DATA_TRUNCATED，*length 是数据值未被截短之前的长度，其实际长度是 *length 和 buffer_length 两者当中比较小的那个。
- my_bool *is_null
  这个指针指向的变量用来表明此数据值是不是对应着一个 NULL 值。
  在用于输入时，应该根据将被发送到服务器去的值是 NULL 还是 NOT NULL 而把这个指针所指向的变量设置为“真”或“假”。特例：如果绑定到这个参数的值不可能是 NULL，你可以把 is_null 直接设置为零而不是 my_bool 类型的变量的地址；如果绑定到这个参数的值只能是 NULL，你只要把 buffer_type 设置为 MYSQL_TYPE_NULL 就足够了，其他 MYSQL_BIND 成员如何设置都无关紧要。
  在用于输出时，应该根据将从服务器接收的值是 NULL 还是 NOT NULL 而把这个指针所指向的变量设置为“真”或“假”。

- my_bool is_unsigned
  这个标志用来表明 buffer 所指向的变量是不是一个 unsigned 型 C 变量，不区分输入或输出用途。这个成员只适用于可以加上 unsigned 限定符的 C 数据类型（char 和各种整数类型）。is_unsigned 成员描述的对象是被绑定到这个 MYSQL_BIND 结构的 C 变量，和服务器端的 SQL 值没有任何关系。客户库将根据 is_unsigned 去判断是否需要在 C 值和 SQL 值之间进行符号转换。
- my_bool *error
  在用于输出时，这个指针指向的变量可以告诉我们从服务器取回的值是不是被截短了。在取回一个数据行后，如果刚才没有发生错误，这个指针指向的变量的值将是“假”；如果因为某个数值超出了取值范围或是某个字符串值太长等原因导致数据被截短了，该变量的值将是“真”。这种截短检查是默认启用的，但可以通过使用 MYSQL_REPORT_DATA_TRUNCTION 选项调用 mysql_options()函数的办法加以控制。
  error 成员是从 MySQL 5.0.3 版开始引入的。
  表 G-4 列出了在使用 C 语言变量向服务器发送数据时与各种 C 变量数据类型相对应的 buffer_type 值。如果那个变量是 unsigned 型的，你还应该把 is_unsigned 值设置为“真”。如果 C 变量的类型和服务器端的 SQL 值的数据类型在表 G-4 里出现在了同一行上，输入数据无须转换就可以使用。比如说，如果你是使用一个 short int 变量向一个 SAMLLINT 数据列提供数据，就不需要进行任何转换；如果是用一个 short int 变量向一个 DECIMAL 数据列提供数据，就需要进行转换。

表G-4　**MYSQL_BIND**结构用于输入时的**buffer_type**值

C输入变量的类型	**buffer_type**值	兼容的SQL数据类型
signed char	MYSQL_TYPE_TINY	TINYINT
short int	MYSQL_TYPE_SHORT	SMALLINT
int	MYSQL_TYPE_LONG	INT
long long int	MYSQL_TYPE_LONGLONG	BIGINT
float	MYSQL_TYPE_FLOAT	FLOAT
double	MYSQL_TYPE_DOUBLE	DOUBLE
MYSQL_TIME	MYSQL_TYPE_TIME	TIME
MYSQL_TIME	MYSQL_TYPE_DATE	DATE
MYSQL_TIME	MYSQL_TYPE_DATETIME	DATETIME
MYSQL_TIME	MYSQL_TYPE_TIMESTAMP	TIMESTAMP
char[]	MYSQL_TYPE_STRING	TEXT, CHAR, VARCHAR
char[]	MYSQL_TYPE_BLOB	BLOB, BINARY, VARBINARY
	MYSQL_TYPE_NULL	NULL

MYSQL_TYPE_STRING 和 MYSQL_TYPE_BLOB 值分别用于非二进制字符串和二进制字符串。

MYSQL_TYPE_NULL 值只适用于某个输入参数永远是 NULL 的情况。在所有其他的情况下，不仅应该把 MYSQL_BIND 结构的 buffer_type 成员设置为其他 MYSQL_TYPE_XXX 值之一，还应该在执行一条语句之前根据其输入参数是否为 NULL 而对相应的 in_null 成员作出正确的设置。

表 G-5 列出了在使用 C 语言变量从服务器接收数据时与各种 C 变量数据类型相对应的 buffer_type 值。如果那个变量是 unsigned 型的，还应该把 is_unsigned 值设置为“真”。如果用来检索数据值的 C 变量的类型和服务器端的 SQL 值的数据类型在表 G-5 里出现在了同一行上，接收自服务器的 SQL 值无须转换就可以使用。比如说，如果你是把一个 SAMLLINT 数据列值取回到一个

short int 变量，就不需要进行任何转换；如果你是把它取回到一个 char[]变量，该值将被转换为字符串形式。

表G-5 **MYSQL_BIND**结构用于输出时的**buffer_type**值

SQL源数据的类型	**buffer_type**值	兼容的C变量类型
TINYINT	MYSQL_TYPE_TINY	signed char
SMALLINT	MYSQL_TYPE_SHORT	short int
MEDIUMINT	MYSQL_TYPE_INT24	int
INT	MYSQL_TYPE_LONG	int
BIGINT	MYSQL_TYPE_LONGLONG	long long int
FLOAT	MYSQL_TYPE_FLOAT	float
DOUBLE	MYSQL_TYPE_DOUBLE	double
DECIMAL	MYSQL_TYPE_NEWDECIMAL	char[]
YEAR	MYSQL_TYPE_SHORT	short int
TIME	MYSQL_TYPE_TIME	MYSQL_TIME
DATE	MYSQL_TYPE_DATE	MYSQL_TIME
DATETIME	MYSQL_TYPE_DATETIME	MYSQL_TIME
TIMESTAMP	MYSQL_TYPE_TIMESTAMP	MYSQL_TIME
CHAR, BINARY	MYSQL_TYPE_STRING	char[]
VARCHAR, VARBINARY	MYSQL_TYPE_VAR_STRING	char[]
TINYBLOB, TINYTEXT	MYSQL_TYPE_TINY_BLOB	char[]
BLOB, TEXT	MYSQL_TYPE_BLOB	char[]
MEDIUMBLOB, MEDIUMTEXT	MYSQL_TYPE_MEDIUM_BLOB	char[]
LONGBLOB, LONGTEXT	MYSQL_TYPE_LONG_BLOB	char[]
BIT	MYSQL_TYPE_BIT	char[]

DECIMAL值和BIT值将默认返回为字符串。如果你给出了一个char[]变量来接收一个DECIMAL值，你得到的将是该数值的字符串表示形式。如果你使用的是一个数值型变量，字符串将被转换为数值形式。如果需要把一个 BIT 值接收为一个数值，就应该在你的查询里把它转换为数值形式（例如：SLELECT my_bit_value+0 ...）并把一个整数变量绑定到相应的 MYSQL_BIND 结构上。

为了区分结果集里的某个输出列是一个非二进制字符串输出列还是一个二进制字符串输出列，应该使用 mysql_stmt_result_metadata()函数来获得结果集的元数据并检查对应于该输出列的 charsetnr 成员。如果该 charsetnr 成员的值是 63，则表明那是一个二进制字符串，任何其他值都表明那是一个非二进制字符串。

- MYSQL_TIME

  这个结构用来向服务器发送日期/时间值和从服务器接收它们。这就需要把一个 MYSQL_TIME 结构和一个 MYSQL_BIND 结构关联在一起，具体做法是把 MYSQL_BIND 结构中的 buffer 成员设置为一个 MYSQL_TIME 变量的地址。

  MYSQL_TIME 结构适用于 DATETIME、TIMESTAMP、DATE 和 TIME 类型，但不适用于某种给定类型的结构成员将被忽略。比如说，month、year 和 day 成员不适用于 TIME 值，而 hour、minute 和 second 成员不适用于 DATE 值。

  MYSQL_TIME 结构包含许多个成员，但它们当中只有一部分允许从外部访问。如下所示的定义列出了 MYSQL_TIME 结构里允许从外部访问的成员：

```
typedef struct st_mysql_time
{
```

```
    unsigned int  year;
    unsigned int  month;
    unsigned int  day;
    unsigned int  hour;
    unsigned int  minute;
    unsigned int  second;
    unsigned long second_part;
    my_bool       neg;
    ...
  } MYSQL_TIME
```

这些成员的含义如下。

- year、month、day
  有日期部分的日期/时间值里的年、月、日部分。
- hour、minute、second、second_part
  有时间部分的日期/时间值里的时、分、秒、小数秒部分。
- neg
  这个标志用来表明这个 MYSQL_TIME 结构里包含的日期/时间值是不是一个负数。

G.2.3　操作符形式的宏

为了让我们能够更方便地测试 MYSQL_FIELD 结构中的成员，mysql.h 文件还定义了一些宏。IS_NUM()对 type 成员进行测试，其他几个宏则对 flags 成员进行测试。

- IS_NUM()。如果数据列是数值类型，则为真（非零）：

```
if (IS_NUM (field->type))
  printf ("Field %s is numeric\n", field->name);
```

- IS_PRI_KEY()。如果数据列是某个 PRIMARY KEY 的组成部分，则为真（非零）：

```
if (IS_PRI_KEY (field->flags))
  printf ("Field %s is part of primary key\n", field->name);
```

- IS_NOT_NULL()。如果数据列不允许包含 NULL 值，则为真（非零）：

```
if (IS_NOT_NULL (field->flags))
  printf ("Field %s values cannot be NULL\n", field->name);
```

- IS_BLOB()。如果数据列是 BLOB 或 TEXT 类型，则为真（非零）。这个宏是通过测试 flags 成员的 BLOB_FLAGS 位而得出结论的，但因为 BLOB_FLAGS 位已逐渐被淘汰，所以使用 IS_BLOB()宏的做法现在已经很少见了。

G.3　C API 函数

以下各节将对客户端库所提供的 C API 函数按功能分类并作出详细的介绍，各类别中的函数将按字母表顺序排列。有些参数会在后面的讨论内容里反复出现，先在此对它们的含义做一个集中的说明。

- conn。一个指针，它指向与某服务器连接相关联的 MYSQL 连接处理程序。
- res_set。一个指针，它指向 MYSQL_RES 结果集结构。
- field。一个指针，它指向一个 MYSQL_FIELD 数据列信息结构。
- row。一个 MYSQL_ROW 数组，即结果集中的某个数据行。
- row_num。结果集里的数据行的序号，从 0 到 mysql_num_row()-1（数据行总个数减去 1）。

- col_num。结果集里某一个数据行的数据列的序号，从 0 到 mysql_num_fields()-1（数据列总个数减去 1）。
- stmt。用于预处理语句的处理程序。

为简洁起见，如果后面的讨论内容在提到以上参数的时候没有对之作出特别的说明，大家就可以把它们理解为上述含义。

G.3.1 客户库初始化和收尾处理例程

在这一节里，我们将讨论对 C API 库进行初始化和收尾处理的例程。这样的库其实有两个，但因为它们的调用接口完全一样，所以它们在程序的源代码里体现不出任何区别。一个给定的程序在运行时将使用的客户库取决于它的二进制可执行映像是通过链接哪个客户库而生成的。

- 通过链接 libmysqlcilent 库而生成的程序将用来访问独立 MySQL 服务器。
- 通过链接 libmysqld 库而生成的程序将包含嵌入式服务器。

只要在程序里正确地使用了 mysql_library_init()和 mysql_librayr_end()函数对客户库进行初始化和收尾处理，就可以用同样的源代码生成一个可以用来访问独立服务器的客户程序，或是生成一个自身包含嵌入式服务器的一体化应用程序，而这完全取决于你在链接阶段选择的是哪一个客户库。关于如何选择 C API 库以及具体链接步骤的讨论见 G.1 节。

- void
  mysql_library_end (void);

  对客户库进行收尾处理。应该在结束与服务器的通信之后才调用这个函数。如果程序使用的是嵌入式服务器库，本例程将关闭嵌入式服务器。

  这个例程是从 MySQL 5.0.3 版开始引入的。在 5.0.3 版之前，你可以调用 mysql_server_end()函数。

- int
  mysql_library_init (int argc, char **argv, char **groups);

  对客户库进行初始化。如果成功，返回零；否则，返回非零值。在调用任何其他 mysql-xxx()函数之前必须先调用这个函数。如果程序使用的是嵌入式服务器库，本例程将对嵌入式服务器进行初始化。

  如果程序使用了一个嵌入式服务器，argc 和 argv 参数的用法将和在 C 程序里传递给 main()函数的标准化参数一样：argc 是参数的个数；如果没有任何参数，argc 应该是零。否则，argc 将是实际传递给服务器的参数的个数。argv 是一个以空字符作为字符串结束标记的字符串数组，其中包含着所有的参数。请注意，argv[0]将被忽略。

  groups 参数是一个以空字符为字符串结束标记的字符串数组，它用来给出嵌入式服务器在启动时应该从选项文件里读取的选项组。这个数组的最后一个元素应该是 NULL。如果 groups 本身是 NULL，服务器将默认读取[server]和[embedded]选项组。在 groups 数组里列出的选项组的名字不应该用“[”和“]”字符括起来。

  这个例程是从 MySQL 5.0.3 版开始引入的。在 5.0.3 版之前，你可以调用 mysql_server_init()函数。

- void
  mysql_server_end (void);

这个例程是 mysql_library_end()函数的一个同义词，但可以用在 MySQL 5.0.3 之前的版本里。

- int
  mysql_server_init (int argc, char **argv, char **groups);

这个例程是 mysql_library_init()函数的一个同义词,但可以用在 MySQL 5.0.3 之前的版本里。

G.3.2　连接管理类函数

连接管理类函数能完成以下几项工作：（1）建立或断开与服务器的连接；（2）对与此有关的各种连接选项进行设置；（3）在连接因超时而自动切断时重建连接；（4）改变当前用户名或字符集等连接设置。

在实际工作中，最常见的做法是这样的：先调用 mysql_init()函数去初始化一个连接处理程序；然后把这个连接处理程序传递给 mysql_real_connect()函数去建立一个连接；最后，当你不再需要用到这个连接的时候，调用 mysql_close()函数来关闭之。如果还想对连接选项做些特殊的设置或者需要建立的是一条 SSL 连接，就得在调用 mysql_init()函数之后、调用 mysql_real_connect()函数之前发出一些 mysql_options()或 mysql_ssl_set()调用。

- my_bool
  ```
  mysql_change_user (MYSQL *conn,
                     const char *user_name,
                     const char *password,
                     const char *db_name);
  ```
  在 conn 参数给定的连接上改变当前用户和默认数据库。如果你在用到某个数据表的时候没有指明它来自哪一个数据库，系统就将默认它来自当前的默认数据库。如果 db_name 参数的值是 NULL，则不选定默认数据库。

  mysql_change_user()的返回值是布尔类型，它返回真（非零）值的条件是：（1）用户有权连接服务器；（2）如果还给出了 db_name 参数，该用户还必须有权访问给定的数据库。对于其他情况，这个函数将返回假（零）值，当前用户和默认数据库不会改变。

  与先关闭当前连接然后再使用另一套参数来重新打开一个新连接的做法相比，用这个函数来改变当前用户和默认数据库的做法要快一些。你还可以用这个函数来实现永久性连接（persistent connection），使你的客户程序在执行期间能够为多名用户提供服务。

- void
  mysql_close (MYSQL *conn);

  关闭 conn 参数给定的连接。调用这个函数将结束你的客户程序与服务器之间的这次会话。如果连接处理程序是由 mysql_init()函数自动分配的，mysql_close()将释放之。

  如果与服务器的连接没有成功，不要调用 mysql_close()。但如果 mysql_init()分配了处理程序，你可能需要这么做，这样你就可以释放它。

- void
  ```
  mysql_get_character_set_info (MYSQL *conn,
                                MY_CHARSET_INFO *cs_info);
  ```
  检索关于当前客户的客户级字符集的信息。检索结果将保存在 cs_info 所指向的 MY_CHARSET_INFO 结构里。MY_CHARSET_INFO 结构的定义如下：

```
typedef struct character_set
{
  unsigned int      number;     /* character set number            */
  unsigned int      state;      /* character set state             */
  const char        *csname;    /* collation name                  */
  const char        *name;      /* character set name              */
  const char        *comment;   /* comment                         */
  const char        *dir;       /* character set directory         */
  unsigned int      mbminlen;   /* min. length for multibyte strings */
  unsigned int      mbmaxlen;   /* max. length for multibyte strings */
} MY_CHARSET_INFO;
```

- ❑ const char *
  mysql_get_ssl_cipher (MYSQL *conn);

  返回一个以空字符为结束标记的字符串，其内容是为连接而使用的 SSL 算法的名字；如果没有使用任何 SSL 算法，这个字符串将是 NULL。
  这个例程是从 MySQL 5.0.23/5.1.11 版开始引入的。

- ❑ MYSQL *
  mysql_init (MYSQL *conn);

  初始化一个连接处理程序并返回一个指向它的指针。如果 conn 参数指向一个已经存在的 MYSQL 处理程序结构，mysql_init()将对它进行初始化并返回它的地址，如下所示：

```
MYSQL conn_struct, *conn;
conn = mysql_init (&conn_struct);
```

  如果 conn 参数是 NULL，mysql_init()将分配一个新处理程序、初始化之再返回它的地址，如下所示：

```
MYSQL *conn;
conn = mysql_init (NULL);
```

  如有可能，应该尽量使用第二个办法，少用第一个办法。把连接处理程序的资源分配和初始化工作都交给客户库去完成有很多好处。比如说，如果你对 MySQL 软件进行了升级而新版本中的 MYSQL 结构与老版本中的有所差异，采用第二个办法就能使你避免很多与共享库有关的问题。
  如果 mysql_init()调用失败，它将返回 NULL，比如当系统资源不足以让它分配一个新处理程序时。
  如果 mysql_init()调用分配了连接处理程序，mysql_close()调用将在你关闭这条连接时自动回收之。

- ❑ Int
  mysql_options (MYSQL *conn,
                enum mysql_option option,
                const void *arg);

  这个函数能够对 mysql_real_connect()函数的连接建立行为做进一步的调控。应该在调用了 mysql_init()函数之后、在调用 mysql_real_connect()函数之前去调用这个函数。如果需要设置多个选项，可以多次调用 mysql_options()函数。（如果你多次调用 mysql_options()函数去设置同一个选项，mysql_real_connect()函数在建立连接时使用的将是你最后一次设置的值。）

option 参数是你准备设置的连接选项的名字。在设置某个选项的时候，如果还需要用到其他的信息，就必须把它们放在 arg 参数里。arg 参数通常被解释为一个指针。）如果不需要用到其他信息，就必须把 arg 参数值设置为 NULL。（在 MySQL 5.1.18 之前，arg 声明为 const char* 而不是 const void*。）

如果选项设置成功，mysql_options() 函数将返回零；如果 option 参数值不合法，则返回非零。option 参数目前有以下几种可取值。如果选用 libmysql client 而非 libmysqld 与客户程序连接，那些被指定应用于嵌入式服务器的选项将被忽略。

- MYSQL_INIT_COMMAND

  在连接成功后立刻执行一条查询命令。arg 指向一个以空字符结尾的字符串，字符串的内容就是你准备执行的语句。这条语句在这个连接每次重建成功后（比如你调用了 mysql_ping() 函数的时候）也会立刻执行。这个查询所返回的结果集将被丢弃。

- MYSQL_OPT_COMPRESS

  如果客户和服务器都支持，本次连接将使用压缩的客户/服务器协议。这个函数的 arg 参数是 NULL。

  这个选项也可以在你调用 mysql_real_connect() 函数时进行设定。

- MYSQL_OPT_CONNECT_TIMEOUT

  连接动作的倒计时时间：如果在经过这么多秒之后仍未成功地连接上服务器，客户程序将放弃这次执行。这个函数的 arg 参数（指针）指向一个 unsigned int 值，其中存放着倒计时的时间值。

- MYSQL_OPT_GUESS_CONNECTION

  如果程序里包含一个嵌入式服务器，这个选项将允许服务器库选择是使用嵌入式服务器库、还是使用一个远程服务器。如果主机名被设置为一个不是 localhost 的值，它“猜测”的结果是使用一个远程服务器。arg 参数应该是 NULL。

  这种“猜测”是默认启用的。MYSQL_OPT_USE_EMBEDDED_CONNECTION 或 MYSQL_OPT_USE_REMOTE_CONNECTION 选项可以用来强制进行指定类型的连接。

- MYSQL_OPT_LOCAL_INFILE

  启用或者禁用 LOAD DATA LOCAL 语句。这个函数的 arg 参数应该是 NULL，表示禁用此功能；或者是指向一个 unsigned int 值的指针：如果这个值非零，则启用此项功能；如果这个值是零，则禁用之。但如果服务器被配置成不允许使用 LOAD DATA LOCAL 语句的情况，这个选项将没有任何效果。

- MYSQL_OPT_NAMED_PIPE

  要求客户程序使用命名管道来连接服务器。这个函数的 arg 参数是 NULL。这个选项是为将在 Windows 系统上使用的客户程序而准备的，而且只能用来连接基于 Windows NT 的 MySQL 服务器。

- MYSQL_OPT_PROTOCOL

  给出在连接服务器时使用的协议（需要假设服务器支持该协议）。arg 参数应该指向一个包含着协议代码的 unisgned int 值。允许使用的协议代码包括 MYSQL_PROTOCOL_MEMOPRY（共享内存）、MYSQL_PROTOCOL_PIPE（Windows 命名管道）、MYSQL_PROTOCOL_SOCKET（Unix 套接字）和 MYSQL_PROTOCOL_TCP（TCP/IP）。

- MYSQL_OPT_READ_TIMEOUT

从服务器读取数据时的倒计时等待时间，以秒为计算单位。这个选项只适用于 TCP/IP 连接，并且在 MySQL 5.0.25/5.1.12 版之前只适用于 Windows 平台。arg 参数应该是一个指针并且应该指向一个包含着倒计时秒数的 unisgned int 值。因为在首次读取失败的情况下有可能进行多次重试，所以实际生效的倒计时等待时间是这个选项值的 3 倍。

- MYSQL_OPT_RECONNECT

  启用或禁用对非正常断开的连接进行自动重建的行为。arg 参数应该指向一个被设置为“真”或“假”的 my_bool 值。

  断线续接功能在 MySQL 5.0.3 及以后的版本里是默认启用的。为了更细致地对断线续接行为进行调控，MySQL 从 5.0.13 版开始引入了这个选项。

- MYSQL_OPT_SET_CLIENT_IP

  如果程序包含着一个具备身份验证支持的嵌入式服务器，这个选项将导致该服务器把连接请求视为来自 arg 参数所给定的 IP 地址。arg 参数应该指向一个以空字符为结束标记的数值型 IP 地址字符串（例如："192.168.3.12"）。

- MYSQL_OPT_SSL_VERIFY_SERVER_CERT

  启用或禁用对服务器端信任证书中的“Common Name”字段的验证。该字段的值必须与服务器的主机名匹配，否则连接尝试将失败。这有用助于防止各种“二传手”攻击手段。arg 参数应该指向一个被设置为“真”或“假”的 my_bool 值。这种验证是默认禁用的。

  这个选项是从 MySQL 5.0.23/5.1.11 版开始引入的。

- MYSQL_OPT_USE_EMBEDDED_CONNECTION

  如果程序包含着一个嵌入式服务器，这个选项告诉服务器库应该使用嵌入式服务器库而不是一个远程服务器。arg 参数应该是 NULL。

- MYSQL_OPT_USE_REMOTE_CONNECTION

  如果程序包含一个嵌入式服务器，这个选项告诉服务器库应该使用一个远程服务器而不是嵌入式服务器库。arg 参数应该是 NULL。

- MYSQL_OPT_USE_RESULT

  未使用。

- MYSQL_OPT_WRITE_TIMEOUT

  向服务器写入数据时的倒计时等待时间，以秒为计算单位。这个选项只适用于 TCP/IP 连接，并且在 MySQL 5.0.25/5.1.12 版之前只适用于 Windows 平台。arg 参数应该是一个指针并且指向一个包含倒计时秒数的 unisgned int 值。因为在首次写入失败的情况下有可能进行多次重试，所以实际生效的倒计时等待时间是这个选项值和 net_retry_count 值的乘积。

- MYSQL_READ_DEFAULT_FILE

  要求客户程序从给定的选项文件里读取连接参数，而不使用标准选项文件里的选项。这个函数的 arg 参数（指针）指向一个以空结尾的字符串，其内容是一个文件名，客户程序将从这个文件的[client]选项组里读取连接选项。如果你还使用 MYSQL_READ_ DEFAULT_GROUP 选项给出了一个选项文件组名称，客户程序就还将读取这个选项文件里的指定选项组里的选项。

- MYSQL_READ_DEFAULT_GROUP

  要求客户程序从给定的选项组中读取连接参数。这个函数的 arg 参数（指针）指向一个以空结尾的字符串，其内容是一个选项组名称（注意：选项组名称的两端不需要用“[”和“]”

字符括起来)；客户程序将从选项文件的[client]选项组和本选项给定的选项组里读取连接选项。如果你还使用 MYSQL_READ_DEFAULT_GROUP 选项给出了一个选项文件名，客户程序将只读取该选项文件里选项；否则，客户程序将到一系列标准选项文件里去读取选项。

如果你既没有给出 MYSQL_READ_DEFAULT_FILE 选项，也没有给出 MYSQL_READ_DEFAULT_GROUP 选项，客户程序将不读取任何选项文件。

- MYSQL_REPORT_DATA_TRUNCATION

  这个选项控制着在使用预处理语句相关的二进制协议时，是否通过 MYSQL_BIND 结构的 error 成员报告数据截短错误。arg 参数应该是一个指向一个 my_bool 变量的指针，该变量为零或非零的含义分别是禁用或启用截短报告功能。截短报告功能是默认启用的。

  这个选项是从 MySQL 5.0.3 版开始引入的。

- MYSQL_SECURE_AUTH

  这个选项控制着是否需要使用更安全的身份验证。arg 参数应该是一个指向一个 my_bool 变量的指针。MySQL 4.1 系列版本实现了一种更安全的口令加密算法，arg 参数所指向的变量为零或非零的含义分别是允许或不允许连接到一个不支持该算法的服务器。

- MYSQL_SET_CHARSET_DIR

  给出字符集文件所在子目录的路径名。arg 参数应该指向一个包含那个子目录路径名的字符串，该字符串以空字节为结束标记。这个子目录位于客户主机上；当客户程序需要访问的字符集没被编译到客户库里、但你手里有该字符集的定义文件时，这个选项就派上用场了。

- MYSQL_SET_CHARSET_NAME

  给出将被用作默认字符集的名字。arg 参数应该指向一个包含该字符集名字的字符串，该字符串以空字节为结束标记。

- MYSQL_SHARED_MEMORY_BASE_NAME

  给出用于共享内存连接的共享内存的名字。arg 参数应该指向一个包含该名字的字符串，该字符串以空字节为结束标记。这个选项只能用在 Windows 客户程序里，并且只能用来连接已经启用了共享内存支持的 Windows 服务器。

在为设置 MYSQL_READ_DEFAULT_FILE 或者 MYSQL_SET_CHARSET_DIR 等选项而给出的 Windows 路径名里，把“\”字符写成“/”或“\\”都是可以的。

使用 mysql_options()函数设置 MYSQL_READ_DEFAULT_FILE 或 MYSQL_READ_DEFAULT_GROUP 选项将导致 mysql_real_connect()函数去读取选项文件。mysql_real_connect()函数可以识别以下选项：

```
character-sets-dir=charset_directory_path
compress
connect-timeout=seconds
database=db_name
debug
default-character-set=charset_name
disable-local-infile
host=host_name
init-command=stmt
interactive-timeout=seconds
local-infile[={0|1}]
max-allowed-packet=size
multi-queries
multi-results
```

```
multi-statements
password=your_pass
pipe
port=port_num
protocol=protocol_type
report-data-truncation
return-found-rows
secure-auth
shared-memory-base-name=name
socket=socket_name
ssl-ca=file_name
ssl-capath=dir_name
ssl-cert=file_name
ssl-cipher=str
ssl-key=file_name
timeout=seconds
user=user_name
```

如果传递给 mysql_real_connect()函数的 host、user、password、database、port 或 socket 参数的值不是 NULL，从选项文件里找到的同名选项将被前者覆盖而不起任何作用。

给出 multi-results 选项的效果等同于在 mysql_real_connect()函数的 flags 参数值里包含 CLIENT_MULTI_RESULTS。给出 multi-queries 或 multi-statement 选项的效果等同于在 mysql_real_connect()函数的 flags 参数值里包含 CLIENT_MULTI_STATEMENTS（这还将同时启用 CLIENT_MULTI_RESULTS）。

timeout 选项目前仍可以被识别，但已经被淘汰，请用 connect-timeout 选项代替。

在下面的例子里，我们用一组 mysql_options()函数调用对连接选项进行了设置，这些设置的效果是：mysql_real_connect()函数将从 C:\my.ini.extra 文件的[client]和[mygroup]选项组读取信息、使用一个命名管道和一段 10 秒钟的倒计时等待时间去建立连接、在连接建立后立刻执行一条 SET NAMES 'utf8'语句。

```
MYSQL *conn;
unsigned int timeout;

if ((conn = mysql_init (NULL)) == NULL)
  ... deal with error ...
mysql_options (conn, MYSQL_READ_DEFAULT_FILE, "C:/my.ini.extra");
mysql_options (conn, MYSQL_READ_DEFAULT_GROUP, "mygroup");
mysql_options (conn, MYSQL_OPT_NAMED_PIPE, NULL);
timeout = 10;
mysql_options (conn, MYSQL_OPT_CONNECT_TIMEOUT, (char *) &timeout);
mysql_options (conn, MYSQL_INIT_COMMAND, "SET NAMES 'utf8'");
if (mysql_real_connect (conn, ...) == NULL)
   ... deal with error ...
```

- int

  mysql_ping (MYSQL *conn);

  检查 conn 参数所给定的连接是否仍然有效，如果已经断开，mysql_ping()将使用客户程序当初建立这个连接时使用的选项去重建这个连接。因此，如果最初的 mysql_real_connect()调用没有成功，就不应该再发出 mysql_ping()调用。如果该连接依然有效或者成功地再次建立起这个连接，mysql_ping ()将返回零；如果执行出错，则将返回一个非零值。

- MYSQL *

```
mysql_real_connect (MYSQL *conn,
                    const char *host_name,
                    const char *user_name,
                    const char *password,
                    const char *db_name,
                    unsigned int port_num,
                    const char *socket_name,
                    unsigned long flags);
```

连接服务器并返回一个指向该连接处理程序的指针。这个函数的 conn 参数（指针）必须指向一个已经存在且已经用 mysql_init()初始化过的连接处理程序。如果调用成功，这个函数的返回值将是该处理程序的地址；如果执行出错，则将返回 NULL。

如果连接操作没有成功，你可以把 conn 传递给 mysql_errno()或 mysql_error()函数去获得出错信息；但你不能把它传递给客户库中必须以连接成功为前提的其他例程。

其余参数指定如何连接到服务器。对于设定为 NULL 或零值的参数，其值可以由 mysql_real_connect()读取的选项文件中的选项提供。（客户可以通过 MYSQL_READ_DEFAULT_FILE 或 MYSQL_READ_DEFAULT_GROUP 选项调用 mysql_options()，让 mysql_real_connect()读取选项文件。）

host_name 参数负责设定 MySQL 服务器主机的主机名。表 G-6 列出了将在 Unix 和 Windows 系统上运行的客户程序用于各种 host_name 参数值的连接协议。主机名"hostname"只能在基于 Unix 的系统上使用，它表示你想通过一个 Unix 套接字而不是通过一条 TCP/IP 路径来建立连接。如果想使用 TCP/IP 来连接一个运行在本地主机上运行的服务器，就需要把 host_name 参数的值设置为"127.0.0.1"(即以字符串形式表示的本地主机的自连接环路接口的 IP 地址）而不能把它设置为"localhost"。

表G-6　服务器**hostname**参数类型设定客户连接协议

hostname参数值	Unix连接协议	Windows连接协议
主机名	使用TCP/IP连接指定主机	使用TCP/IP连接指定主机
IP数字	使用TCP/IP连接指定主机	使用TCP/IP连接指定主机
localhost	使用Unix套接字文件连接本地主机	使用共享内存连接本地主机，如果不成功，则使用TCP/IP连接
127.0.0.1	使用TCP/IP连接本地主机	使用TCP/IP连接本地主机
.（句点）	不适用	使用命名管道连接本地主机
NULL	使用Unix套接字连接本地主机	先尝试使用命名管道连接本地主机，如果不成功，再使用TCP/IP连接

user_name 就是你的 MySQL 用户名。如果这个参数的值是 NULL，客户端库将提供一个默认的名字：在 Unix 系统上，这个默认的用户名就是你的登录名；在 Windows 系统上，它将是环境变量 USER 的值——如果这个环境变量没有定义，则使用"ODBC"作为默认值。

password 参数就是你的口令。如果这个参数的值是 NULL，你就只能连接到 user 权限数据表里与给定主机名和你的用户名相对应且口令字段为空的那个数据库，连接其他数据库将以失败告终。

db_name 参数就是你打算使用的默认数据库的名字。如果这个参数的值是 NULL，则表示客户程序不选定初始的默认数据库。

port_num 参数是 TCP/IP 连接所使用的端口号。如果这个参数的值是 0，则表示客户程序将使

用默认的端口号进行连接。

socket_name 参数是用于连接到"localhost"的 Unix 套接字（如果你使用的是 Unix 系统的话）或命名管道（如果你使用的是 Windows 系统的话）的文件名。如果这个参数的值是 NULL，则表示客户程序将使用默认的套接字或命名管道。

port_num 和 socket_name 参数值的具体含义还要取决于图 G-6 中 host_name 参数的值。

flags 参数的值既可以是下列选项中的一个或者多个，也可以是 0（意思是“没有选项”）。这些选项将对服务器的运行情况产生影响。

- CLIENT_COMPRESS
  如果服务器支持，这条连接将使用压缩的客户/服务器协议。
- CLIENT_FOUND_ROWS
  对于 UPDATE 查询，服务器将返回与该查询的 WHERE 子句相匹配的数据行个数，而不是被它改变的数据行的个数。这个选项将阻碍 MySQL 优化器，使修改命令执行得稍慢一些。
- CLIENT_IGNORE_SIGPIPE
  阻止客户端库为 SIGPIPE 信号安装一个处理程序。这对于已有自己的处理程序的应用而言很有用。
- CLIENT_IGNORE_SPACE
  一般来说，在内建函数的名字后面必须立刻写出参数表的左括号，两者之间不能有空格。这个选项将使服务器忽略函数名和参数表之间的所有空格，而这么做的副作用是让所有的函数名变成保留字。
- CLIENT_INTERACTIVE
  表明这个客户程序是一个交互式客户程序。对于由交互式客户程序建立的连接，如果它在服务器变量 interactive_timeout 所设定的时间（以秒为单位）内没有操作动作，服务器就可以把该连接关闭掉。一般说来，interactive_timeout 变量的值与 wait_timeout 变量的值相等。
- CLIENT_LOCAL_FILES
  允许使用 LOAD DATA LOCAL 语句。但如果 MySQL 服务器已被配置成不允许使用 LOAD DATA LOCAL 语句的话，这个选项将没有任何效果。
- CLIENT_MULTI_RESULTS
  允许使用 mysql_more_results()和 mysql_next_result()函数取回多个结果集。
  只要你在程序里用 CALL 语句调用过会返回一个结果集的存储过程，你就必须给出这个选项。否则，就会导致一个错误。
- CLIENT_MULTI_STATEMENTS
  允许一次执行多条语句。当这个功能被启用时，你可以用一个字符串向服务器发送多条语句。这个选项还将启用 CLIENT_MULTI_RESULTS 选项以允许取回多个结果集。
- CLIENT_NO_SCHEMA
  禁用 *db_name.tbl_name.col_name* 语法。如果使用了这个选项，服务器将只能识别查询命令中的 *tbl_name.col_name* 或 *col_name* 语法。

上面这些 flag 参数值都是比特值，所以你可以用二进制位操作符“|”或者“+”操作符把它们组合在一起使用以得到叠加的效果。比如说，下面这两个表达式就是等价的：

```
CLIENT_COMPRESS | CLIENT_ODBC
```

CLIENT_COMPRESS + CLIENT_ODBC

mysql_com.h 在前面列出了其他 CLIENT_*XXX* 值，但它们要么已不再使用，要么只用于内部，所以客户程序不应在 flags 中指定它们。

- int

  mysql_select_db (MYSQL *conn, const char *db_name);

  把名为 db_name 的数据库选取为默认数据库，如果你在此后引用某个数据表时没有明确地给出数据库的名字，它将默认来自该数据库。如果你不具备访问该数据库的权限，mysql_select_db()调用将失败。

  mysql_select_db()函数最主要的用途是在连接期间改用另一个默认数据库。一般来说，在调用 mysql_real_connect()函数的时候就应该指定一个默认数据库，那样要比在连接建立后再调用 mysql_select_db()去指定一个默认数据库的做法更快。

  如果 mysql_select_db()函数调用成功，它将返回零；如果调用失败，则返回一个非零值。

- int

  mysql_set_character_set (MYSQL *conn, const char *cs_name);

  为指定连接设置默认字符集（就像执行了一条 SET NAMES 语句一样）。cs_name 指向一个包含该字符集名字的字符串。

  如果 mysql_set_character_set()函数调用成功，它将返回零；如果调用失败，则返回一个非零值。

  这个例程是从 MySQL 5.0.7 版开始引入的。

- my_bool

```
mysql_ssl_set (MYSQL *conn,
               const char *key,
               const char *cert,
               const char *ca,
               const char *capath,
               const char *cipher);
```

  这个函数用在需要建立一条通往 MySQL 服务器的 SSL 安全化连接的场合。如果 SSL 支持机制没有被编译到客户开发库里，mysql_ssl_set()调用将什么也不做；否则，它将设置 mysql_real_ connect()调用建立一条加密连接所需要的参数。（换句话说，如果想建立一条安全化连接，就得先调用 mysql_ssl_set()函数，然后再调用 mysql_real_connect()函数。）

  mysql_ssl_set()函数调用的返回值永远是 0。在 SSL 参数设置阶段发生的错误将导致随后的 mysql_real_connect()报告一个执行出错。

  mysql_ssl_set()函数的参数 key 是密钥文件的路径名，cert 是证书文件的路径名，ca 是颁证机构文件的路径名。capath 是用来存放信任证书的子目录的路径名，客户主机将使用这个子目录里的证书来验证此后接收到的证书。cipher 是一个字符串，其内容是客户主机使用的一个或者多个加密算法的名称。上述参数的值都允许是 NULL，意思是不需要使用。

  关于如何编写能使用安全连接的客户程序的示例，请参见 7.6 节。

  mysql_ssl_set()函数要求你必须事先对 MySQL 软件做好相应的配置。参见 13.3 节了解必需的背景知识。

G.3.3　出错报告类函数

客户程序需要通过以下函数来查知和报告有关函数调用执行出错的原因。errmsg.h、mysqld_error.h

和 sql_state.h MySQL 头文件中列出了可能的错误代码和消息。

- unsigned int

  mysql_errno (MYSQL *conn);

  返回最近一次调用的客户端库函数（返回状态信息）的出错代码。如果没有出错，则返回 0；否则，将返回一个非零值。

```
if (mysql_errno (conn) == 0)
  printf ("Everything is okay\n");
else
  printf ("Something is wrong!\n");
```

- const char *

  mysql_error (MYSQL *conn);

  返回最近一次调用的客户端库函数(返回状态信息)的出错信息字符串,这个字符串以空结尾。如果没有出错，则将返回一个空字符串（注意：空字符串是一个零长度的字符串""，而不是一个 NULL 指针）。比较常见的做法是先检查前一个函数调用是否执行出错，如果出错，再调用 mysql_error()函数去查明其具体原因；但 mysql_error()函数的返回值本身也可以用来检测前一个函数调用是否执行出错，如下所示：

```
const char *err = mysql_error (conn);
if (err[0] == '\0')                    /* empty string? */
  printf ("Everything is okay\n");
else
  printf ("Something is wrong!\n");
```

- const char *

  mysql_sqlstate (MYSQL *conn);

  返回包含一个 SQLSTATE 错误代码的以空结尾的字符串，这个错误代码是最近调用的返回一个状态信息的客户例程产生的。这个代码是一个包含 5 个字符的字符串。SQLSTATE 值来自 ANSI SQL 和 ODBC 标准。值"0000"意味着没有错误。值"HY000"意味着一般性错误。这个值用于那些还没有分配更具体的 SQLSTATE 代码的 MySQL 错误。

```
if (strcmp (mysql_sqlstate (conn), "00000") == 0)
  printf ("Everything is okay\n");
else
  printf ("Something is wrong!\n");
```

G.3.4　查询构造与执行类函数

客户程序需要通过以下函数来将 SQL 语句发送到服务器去执行。如果查询命令里包含有需要特殊对待的字符，还要用 mysql_hex_string()和 mysql_real_escape_string()对它们进行转义。每个查询命令字符串只能由一条 SQL 语句构成且不得以分号（;）或“\g”结尾，除非启用了 G.3.8 节中的多语句执行功能。分号（;）或“\g”是 mysql 客户程序使用的记号方法，不适用于 MySQL C 客户开发库。

- unsigned long

  mysql_hex_string (char *to_str,
                   const char *from_str,
                   unsigned long from_len);

  对可能包含特殊字符的字符串进行编码，以使它能够在 SQL 语句里使用。

  将被编码的缓冲区以一个计数型字符串的形式给出。from_str 是一个指向该缓冲区的指针，from_len 是该缓冲区的字节长度。mysql_hex_string()函数将把该缓冲区里的每一个字符

编码为两位十六进制数字，并把编码结果写入 to_str 指向的缓冲区，然后加上一个空字节作为结束标记。to_str 必须指向一个已经存在的缓冲区，并且其长度至少要有(from_len*2)+1 个字节。mysql_hex_string()函数的返回值是字符串编码结果的长度，用来充当结束标记的空字节不计算在内。

我们来看一个例子：

```
to_len = mysql_hex_string (to_str, "\0\\\'\"\n\r\032", 7);
printf ("to_len = %d, to_str = %s\n", to_len, to_str);
```

这个例子将产生如下所示的输出：

```
to_len = 14, to_str = 005C27220A0D1A
```

mysql_hex_string()函数所返回的编码字符串本身不包含任何空字节，但以空字节为结束标记，所以完全可以把它用在诸如 strlen()或 strcat()之类的函数里。需要提醒大家的是，这个返回值本身对 SQL 语句而言还不是一个合法的十六进制常数，你必须在它的开头加上"0x"或者是在它的首、尾分别加上"X'"和"'"才能使之成为一个合法的常数。

- int

  mysql_query (MYSQL *conn, const char *stmt_str);

  把给定的以空结尾的字符串发送给服务器去执行。这个字符串不应该包含有二进制数据；具体地说，就是不应该包含有零字节，因为 mysql_query()函数将把它遇到的第一个零字节解释为该查询命令的结束标志。如果你的查询命令真的包含有二进制数据，就应该用 mysql_real_query()函数来发送它。mysql_real_query()的执行速度要比 mysql_query()稍快一些。

  如果执行成功，mysql_query()将返回零；如果执行出错，则将返回一个非零值。所谓“成功的”查询指的是本身没有语法错误且在服务器上执行时也没出错的查询，“成功的”查询并不意味着肯定会有数据行受到它的影响或者会返回一些数据行。

- unsigned long

  ```
  mysql_real_escape_string (MYSQL *conn,
                            char *to_str,
                            const char *from_str,
                            unsigned long from_len);
  ```

  对包含有特殊字符的字符串进行编码，使它能够用在一条 SQL 语句里，编码操作会把当前字符集的因素也考虑在内。所谓的特殊字符以及它们的编码结果见表 G-5。(注意：SQL 匹配模式字符“%”和“_”并没有出现在这份清单里。)

表G-7　**mysql_real_escape_string()**字符编码方案

特殊字符	编码结果
NUL（零字节）	\0（反斜线-0）
反斜线	\\（反斜线-反斜线）
单引号	\'（反斜线-单引号）
双引号	\"（反斜线-双引号）
换行符	\n（反斜线-n）
回车符	\r（反斜线-r）
Ctrl-Z	\Z（反斜线-Z）

MySQL 本身载字符串中需要转义的字符是反斜线和引号字符（如'或"）。mysql_real_escape_string()将其他字符转义为日志文件中更易读和更易处理的字符串。

将被编码的字符串保存在指针参数 from_str 所指向的缓冲区里，这个字符串将被视为一个计数型字符串，其长度（以字节为单位计算）由参数 from_len 给定。mysql_real_escape_string()将把编码结果写到指针参数 to_str 指定的缓冲区里并在其末尾加上一个空字符。指针参数 to_str 所指向的缓冲区至少要有 from_len*2+1 个字节长。在最坏的情况下，from_str 中的每一个字符都需要被编码为一个由两个字符组成的序列，并且还需要为充当结果字符串结束标志的零字节留出一个位置。

mysql_real_escape_string()函数的返回值是编码结果字符串的长度，作为其结束标志的零字节不计算在内。

编码结果字符串的内部没有 NUL 字符，但以零字节结尾，这就使你能够使用 strlen()或 strcat()函数来处理它们。

当你在客户程序里写出字符串的时候，千万不要把 C 语言转义字符（即反斜线字符“\”）本身与你用 mysql_real_escape_string()函数编码出来的转义序列混为一谈。请看下面这段源代码以及它产生的输出：

```
to_len = mysql_real_escape_string (conn, to_str, "\0\\\'\"\n\r\032", 7);
printf ("to_len = %d, to_str = %s\n", to_len, to_str);
```

这段源代码的输出是：

```
to_len = 14, to_str = \0\\\'\"\n\r\Z
```

在上面的例子里，输出字符串 to_str 与源代码中的 mysql_real_escape_string()函数的第三个参数看起来几乎一模一样，但它们在本质上却完全不同。

- ```
  int
  mysql_real_query (MYSQL *conn,
                    const char *stmt_str,
                    unsigned long length);
  ```

  把给定查询命令发送给服务器去执行，查询命令被表示为一个计数型字符串。查询命令的文本由参数 query_str 给定，其长度由参数 length 给定。这种字符串允许包含二进制数据（包括零字节在内）。

  如果执行成功，mysql_real_query()将返回零；如果执行出错，则将返回一个非零值。所谓“成功的”查询指的是本身没有语法错误且在执行时也没出错的查询，“成功的”查询并不意味着肯定会有数据行受到它的影响或者会返回一些数据行。

G.3.5 结果集处理类函数

对于那些会产生结果集的查询，本节里的函数将使你能够检索结果集并访问其中的内容。mysql_store_result()和 mysql_use_result()函数负责创建结果集，在调用本小节中的其他函数之前，你必须先发出一个 mysql_store_result()或 mysql_use_result()调用，表 G-8 对这两个函数进行了对比。

表G-8 **mysql_store_result()**和**mysql_use_result()**的对比

mysql_store_result()	**mysql_use_result()**
结果集中的数据行将由mysql_store_result()亲自取回	mysql_use_result()只对结果集进行初始化，数据行将由后续的mysql_fetch_row()调用取回
  ```

（续）

mysql_store_result()	mysql_use_result()
占用的内存较多，所有的数据行都缓存在客户端	占用的内存较少，每次只存放一个数据行在客户端
慢，因为需要为整个结果集分配内存	快，因为只需为当前数据行分配内存
若mysql_fetch_row()的返回值是NULL，则表明到达结果集里的最后一个数据行，而没出错	若mysql_fetch_row()的返回值是NULL，则表明到达结果集里的最后一个数据行或者发生一个错误，比如因网络通信故障而无法取回当前记录
在发出mysql_store_result()调用之后，你随时都能去调用mysql_num_rows()函数	只有在结果集里的数据行全都被取回之后，mysql_store_result()调用返回正确的数据行计数值
mysql_affected_rows()相当于mysql_num_rows()函数的一个同义词	mysql_affected_rows()无法使用
允许通过mysql_data_seek()、mysql_row_seek()和mysql_row_tell()函数随机访问结果集里的数据行	无法对结果集进行随机访问，你只能依次按照自服务器返回的顺序处理数据行。无法使用mysql_data_seek()、mysql_row_seek()和mysql_row_tell()函数
数据表的锁定时间较短；在数据行被取回之后，锁定就解除了	数据表的锁定时间较长。如果你在此期间挂起了客户程序，数据表将仍处于锁定状态，从而阻塞试图修改数据表的其他客户程序
结果集各MYSQL_FIELD结构中的max_length成员将被设置为有一个有意义的值，即各数据列在结果集里长度最大的那个值的长度	max_length成员无法设置为有一个有意义的值，因为它只有在数据行全都被取回之后才能确定

- my_ulonglong
  mysql_affected_rows (MYSQL *conn);

  返回最近一次 DELETE、INSERT、REPLACE 或 UPDATE 查询所改变的数据行的个数。对于这类查询，应该在 mysql_query() 或 mysql_real_query() 调用成功之后立刻发出一个 mysql_affected_rows()调用。当然，你也可以在发出一个有结果集可供返回的查询命令后调用这个函数。此时，mysql_affected_rows()将与mysql_num_rows()有着同样的行为并受到同样的限制，比如返回值在何种情况下才有意义。如果结果集是你通过发出 mysql_use_result()调用而生成的，那么 mysql_affected_rows()函数将不会返回任何有意义的返回值。

  如果尚未发出任何语句，或者语句为 UPDATE 却没改变任何行，或者语句本该返回一些数据行但没能检索到任何数据行，mysql_affected_rows()的返回值将是零。如果这个返回值大于零，那么，对于 DELETE、INSERT、REPLACE 或 UPDATE 查询，它就是受其影响的数据行的个数；对于会产生结果集的查询，它就是结果集里的数据行的个数。如果这个返回值是-1，则表示执行过程中出错或者你调用 mysql_affected_rows()函数的次序不正确（比如说，你在发出一个会返回数据行的查询命令之后、但在把它的结果集全部检索完毕之前发出了 mysql_affected_rows()调用）。然而，mysql_affected_rows()函数的返回值是一个无符号整数，如果你想判断这个返回值是否对应着一个负整数，就必须先把它转换为一个带符号整数才能进行进行比较操作，如下所示：

```
if ((long) mysql_affected_rows (conn) == -1)
  fprintf (stderr, "Error!\n");
```

  如果你已指定客户程序应返回与 UPDATE 语句匹配的行数，那么，mysql_affected_rows()

就会返回那个行数，而不是被它实际改变的数据行的个数。(如果要修改的数据列与新的值一样,MySQL 将不会更新数据行。)将 flags 参数中的 CLIENT_FOUND_ROWS 传递给 mysql_real_connect()可以选择这种行为。

mysql_real_connect()的返回值是 my_ulonglong 类型。G.2.1 节对如何打印输出这一类型的值的方法进行了介绍。

- void

  mysql_data_seek (MYSQL_RES *res_set, my_ulonglong row_num);

  定位到结果集里的指定数据行处。row_num 参数的取值范围是 0 到 mysql_num_rows(res_set)-1；如果你给出的 row_num 参数值超出了这个范围，执行情况将无法预料。

  mysql_data_seek()函数只有在整个结果集全都被检索到客户主机之后才能正确执行，所以你应该只在结果集是由 mysql_store_result()而不是由 mysql_use_result()所创建的场合里才使用它。

  mysql_data_seek()与 mysql_row_seek()是有区别的,后者的第二个参数却被解释为数据行在结果集里的偏移量（比如 mysql_row_tell()调用的返回值)。

- MYSQL_FIELD *

  mysql_fetch_field (MYSQL_RES *res_set);

  返回一个结构,里面包含着关于结果集里的某个数据列的描述性信息(即该数据列的元数据)。在成功地执行了一条会返回一些数据行的查询命令之后，第一个 mysql_fetch_field()调用将返回关于结果集里的第一个数据列的信息，此后的 mysql_fetch_field()调用将依次返回关于结果集里的下一个数据列的信息，或者——当到达最后一个数据列之后——NULL 值。

  与本函数配合使用的其他函数包括：(1) mysql_field_tell()，用来查知当前数据列是结果集里的第几个数据列；(2) mysql_field_seek()，用来选择下一个 mysql_fetch_field()调用要返回的指定数据列。

  下面这段代码将首先定位到结果集的第一个 MYSQL_FIELD 结构，然后依次提取出各后续数据列的 MYSQL_FIELD 结构：

```
MYSQL_FIELD   *field;
unsigned int  i;

mysql_field_seek (res_set, 0);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  field = mysql_fetch_field (res_set);
  printf ("column %u: name = %s max_length = %lu\n",
          i, field->name, field->max_length);
}
```

- MYSQL_FIELD *

  mysql_fetch_fields (MYSQL_RES *res_set);

  这个函数的返回值是一个数组，数组中的各个元素依次是与结果集里的各个数据列相对应的 MYSQL_FIELD 结构。可以像下面这样来访问这个数组的内容：

```
MYSQL_FIELD   *field;
unsigned int  i;
```

```
field = mysql_fetch_fields (res_set);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  printf ("column %u: name = %s max_length = %lu\n",
          i, field[i].name, field[i].max_length);
}
```

请把这段代码与刚才在介绍 mysql_fetch_field()函数时给出的例子对比。请注意：虽然这两个函数所返回的值是同样的类型，但它们的内容却要用稍微不同的语法去访问。mysql_fetch_field()的返回值（指针）指向一个 MYSQL_FIELD 结构；而 mysql_fetch_fields()的返回值（指针）指向一个以 MYSQL_FIELD 结构为元素的数组。

- MYSQL_FIELD *
  mysql_fetch_field_direct (MYSQL_RES *res_set, unsigned int col_num);

  给定一个某数据列的序号(或者叫下标)，返回对应于该数据列的 MYSQL_FIELD 结构。col_num 参数值的取值范围是 0 到 mysql_num_fields(res_set)-1；如果你给出的 col_num 参数值超出了这个范围，执行情况将无法预料。
  下面是一段直接访问 MYSQL_FIELD 结构的示例代码：

```
MYSQL_FIELD   *field;
unsigned int  i;

for (i = 0; i < mysql_num_fields (res_set); i++)
{
  field = mysql_fetch_field_direct (res_set, i);
  printf ("column %u: name = %s max_length = %lu\n",
          i, field->name, field->max_length);
}
```

- unsigned long *
  mysql_fetch_lengths (MYSQL_RES *res_set);

  返回一个指针，指向一个以 unsigned long 值为元素的数组，元素依次是结果集里的当前数据行的数据列值的长度。每调用一次 mysql_fetch_row()函数，就必须调用一次 mysql_fetch_lengths()；否则，长度值将与数据列值不同步。
  NULL 值的长度是零，但零长度本身并不表明它对应着一个 NULL 值。空字符串的长度也是零，要想区分这两种情况，还必须去检查数据值是否是一个 NULL 指针。
  下面这段代码将把当前数据行的数据列的长度和值打印出来；如果数据列值是 NULL，则打印单词“NULL”：

```
unsigned long *length;

length = mysql_fetch_lengths (res_set);
for (i = 0; i < mysql_num_fields (res_set); i++)
{
  printf ("length is %lu, value is %s\n",
          length[i], (row[i] != NULL ? row[i] : "NULL"));
}
```

- MYSQL_ROW
  mysql_fetch_row (MYSQL_RES *res_set);

  返回一个指针，指针指向结果集里的下一个数据行，数据行被表示为一个字符串数组（取值为 NULL 的数据列是个例外，它们将被表示为 NULL 指针）。数据行的第 *i* 个数据列值就是数组的

第 *i* 个元素，*i* 的取值范围是 0 到 mysql_num_fields(res_set)-1。

各种数据类型（数值类型也不例外）的值都将被表示为一个字符串。如果需要用某个值来进行数学运算，就必须亲自使用 atoi()、atof()、sscanf()等函数去进行类型转换。

如果已经到达结果集里的最后一个数据行，mysql_fetch_row()将返回 NULL 值。（如果结果集是用 mysql_use_result()调用逐行生成的，那么网络通信故障也会使它返回 NULL 值。）

数据值都是以空结尾的，但千万不要把它当做以空结尾的字符串来对待，应该把它当做计数字符串。此时，你需要知道这种数据列的长度，可以通过 mysql_fetch_ lengths()调用来完成这一任务。

请看下面这段代码，它通过一个循环来依次提取构成同一数据行的各项数据并判断它是否是一个 NULL 值：

```
MYSQL_ROW       row;
unsigned int    i;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    printf ("column %u: value is %s\n",
            i, (row[i] == NULL ? "NULL" : "not NULL"));
  }
}
```

数据列值的类型可以通过存放在 MYSQL_FIELD 数据列信息结构里的数据列元数据来确定。可以通过 mysql_fetch_field()、mysql_fetch_fields()、mysql_fetch_field_direct()等调用来完成这一任务。

- unsigned int
  mysql_field_count (MYSQL *conn);

返回给定连接上最近一次查询所选取的数据列的个数。mysql_field_count()函数通常用在 mysql_store_result()或 mysql_use_result()调用返回了一个 NULL 值的场合，它能够告诉你该查询是否应该返回一个结果集：如果返回值是 0，说明该查询的确不会生成一个结果集，其执行过程也就没有出错（例如，对于 INSERT 和 UPDATE）；如果返回的是一个非零值，就说明该查询本应该返回一些数据列，既然不是这样，所以该查询在执行过程中肯定出错了。

下面这段代码演示了如何使用 mysql_field_count()函数来判断是否出错：

```
res_set = mysql_store_result (conn);
if (res_set)      /* a result set was returned */
{
  /* ... process rows here, and then free result set ... */
  mysql_free_result (res_set);
}
else              /* no result set was returned */
{
  /*
   * does the lack of a result set mean that the statement didn't
   * return one, or that it should have but an error occurred?
   */
  if (mysql_field_count (conn) == 0)
  {
    /*
     * statement generated no result set (it was not a SELECT,
```

```
     * SHOW, DESCRIBE, etc.); just report rows-affected value.
     */
    printf ("Number of rows affected: %lu\n",
            (unsigned long) mysql_affected_rows (conn));
  }
  else  /* an error occurred */
  {
    printf ("Problem processing the result set\n");
  }
}
```

- MYSQL_FIELD_OFFSET

  mysql_field_seek (MYSQL_RES *res_set, MYSQL_FIELD_OFFSET offset);

  在结果集里，定位到 offset 参数所指定的那个数据列信息结构，下一个 mysql_fetch_field() 调用将返回该信息结构。offset 参数不是一个某数据列的序号；它是一个 MYSQL_FIELD_OFFSET 类型的值（比如此前的某个 mysql_field_tell() 或 mysql_field_seek() 调用的返回值）。

- MYSQL_FIELD_OFFSET

  mysql_field_tell (MYSQL_RES *res_set);

  返回当前数据列信息结构的偏移量。可以把这个偏移量传递给 mysql_field_seek()。

- void

  mysql_free_result (MYSQL_RES *res_set);

  释放结果集所占用的内存。你必须调用 mysql_free_result() 函数去释放你客户程序所生成的每一个结果集。结果集通常是由 mysql_store_result() 或 mysql_use_result() 调用生成的。对于 mysql_use_result() 调用生成的结果集，mysql_free_result() 将自动取回并丢弃其中尚未取回的数据行。

- my_ulonglong

  mysql_insert_id (MYSQL *conn);

  返回给定连接上最近一次执行的语句所生成的 AUTO_INCREMENT 值。这适用于自动生成的 AUTO_INCREMENT 或保存在列中的文字值。（这异于 LAST_INSERT_ID() SQL 函数，因为后者只返回自动生成的值。）

  如果尚未执行任何查询或者前一个查询根本就不生成一个 AUTO_INCREMENT 值，这个函数调用将返回零。（零返回值与合法的 AUTO_INCREMENT 值是有区别的，因为后者都是些正整数。）

  如果前一个语句出错，mysql_insert_id() 的值是未定义的。

  在发出一个会生成新的 AUTO_INCREMENT 值的语句后，应立刻发出一个 mysql_insert_id() 调用。如果你在发出 mysql_insert_id() 调用之前又执行了另外一条数据库查询命令，返回值就可能会发生变化。请注意：这种行为与 SQL 函数 LAST_INSERT_ID() 是不同的。mysql_insert_id() 由客户端负责处理，每发出一个数据库查询命令，它的返回值就会发生相应的改变；LAST_INSERT_ID() 由服务器端负责处理，其返回值与具体的查询命令相关联，只要你还没有生成一个新的 AUTO_INCREMENT 值，它的返回值就保持不变。

  mysql_insert_id() 函数的返回值与具体的“客户–服务器”连接相关联，不会因发生在其他连接上的 AUTO_INCREMENT 活动而受到影响。

  mysql_insert_id() 的返回值是 my_ulonglong 类型，G.2.1 节对如何打印输出这一类型的值的方法进行了介绍。

- unsigned int
  mysql_num_fields (MYSQL_RES *res_set);

  返回结果集里的数据列的个数。mysql_num_rows()函数的常见用途是对结果集里的当前数据行中的各个数据列进行遍历，如下所示：

```
MYSQL_ROW     row;
unsigned int  i;

while ((row = mysql_fetch_row (res_set)) != NULL)
{
  for (i = 0; i < mysql_num_fields (res_set); i++)
  {
    /* do something with row[i] here ... */
  }
}
```

- my_ulong long
  mysql_num_rows (MYSQL_RES *res_set);

  返回结果集里的数据行的个数。如果结果集是用mysql_store_result()函数生成的，就可以在它调用成功之后的任何时候去发出mysql_num_rows()调用，如下所示：

```
if ((res_set = mysql_store_result (conn)) != NULL)
{
  /* mysql_num_rows() can be called now */
}
```

  如果结果集是用 mysql_use_result()函数生成的，那么，只有在你取回所有数据行之后，mysql_num_rows()调用才会有一个正确的返回值，如下所示：

```
if ((res_set = mysql_use_result (conn)) != NULL)
{
  /* mysql_num_rows() cannot be called yet */
  while ((row = mysql_fetch_row (res_set)) != NULL)
  {
    /* mysql_num_rows() still cannot be called */
  }
  /* mysql_num_rows() can be called now */
}
```

  mysql_num_rows()函数的返回值是my_ulonglong类型，G.2.1节对如何打印输出这一类型的值进行了介绍。

- MYSQL_ROW_OFFSET
  mysql_row_seek (MYSQL_RES *res_set, MYSQL_ROW_OFFSET offset);

  定位到结果集里的指定数据行处。mysql_row_seek()函数与mysql_data_seek()函数的功能相同，但前者的offset值并不是一个某数据行的序号。这个offset参数或者是一个MYSQL_ROW_OFFSET类型的值（比如此前的某个mysql_row_tell()或mysql_row_seek()调用的返回值），或者等于0（意思是定位到结果集里的第一个数据行处）。

  mysql_row_seek()函数的返回值是它这次调用开始执行之前的数据行偏移量。

  mysql_row_seek()函数只有在整个结果集全都被检索到客户主机之后才能正确执行，所以你应该只在结果集是由mysql_store_result()而不是由mysql_use_result()创建时才使用它。

- MYSQL_ROW_OFFSET

  mysql_row_tell (MYSQL_RES *res_set);

  返回结果集里的当前数据行在该结果集中的位置偏移量。这个返回值不是一个数据行序号，可以把它传递给 mysql_row_seek()，但不能把它传递给 mysql_data_seek()。
  mysql_row_tell()函数只有在整个结果集全都被检索到客户主机之后才能正确执行，所以你应该只在结果集是由mysql_store_result()而不是由mysql_use_result()所创建时才使用它。
- MYSQL_RES *

  mysql_store_result (MYSQL *conn);

  在成功地执行了一条语句之后，把该语句的结果集返回并保存到客户主机上。如果查询命令没有数据可供返回或者执行出错，则将返回 NULL，此时，你需要调用 mysql_filed_count()函数或者出错报告类函数，判断它是没有结果集可供返回还是在执行过程中发生了错误。相关示例参见对 mysql_field_count()的描述。
  在完成了对结果集的处理工作后，别忘了用 mysql_free_result()函数去释放它。
  另请参见表 G-8 对 mysql_store_result()和 mysql_use_result()函数的对比。
- MYSQL_RES *

  mysql_use_result (MYSQL *conn);

  在成功地执行了一条数据库查询命令之后，为该查询命令初始化一个结果集，但不把数据行检索到客户主机上。你必须通过 mysql_fetch_row()调用来一个一个地依次取回数据行。如果查询命令没有数据可供返回或者执行出错，则将返回 NULL，此时，你需要调用 mysql_filed_count()函数或者调用出错报告类函数来判断它是没有结果集可供返回，还是在执行过程中发生了错误。相关示例请参见对 mysql_field_count()的描述。
  在完成了对结果集的处理工作后，别忘了用 mysql_free_result()函数去释放它。这是完成语句处理所要做的一切，因为 mysql_free_result()自动取回并丢弃其中尚未被你取回的数据行。
  另请参见表 G-8 对 mysql_store_result()和 mysql_use_result()函数的对比。

G.3.6 信息类函数

这些函数用来收集关于客户程序、服务器、协议版本以及当前连接的信息。这些信息大都是在“客户–服务器”连接建立阶段从服务器上检索出来并被保存到客户程序开发库里的。

- const char *

  mysql_character_set_name (MYSQL *conn);

  返回值：给定连接上的默认字符集的名字。是一个以空结尾的字符串，如"latin1"。
- const char *

  mysql_get_client_info (void);

  返回值：客户程序开发库的版本号。它被表示为一个以空结尾的字符串，比如"5.0.60"。
- unsigned long

  mysql_get_client_version (void);

  返回一个代表着客户库版本的整数。该返回值的格式和 mysql_get_server_version()函数的完全一样。

- `const char *`
  `mysql_get_host_info` `(MYSQL *conn);`

  返回值：关于给定连接的描述性信息。它被表示为一个以空结尾的字符串，比如"Localhost via Unix socket"、"cobra.snake.net via TCP/IP"、". via named pipe"或"Shared memory"。

- `unsigned int`
  `mysql_get_proto_info` `(MYSQL *conn);`

  返回值：一个用来表明给定连接所使用的客户/服务器协议的数字编号。

- `const char *`
  `mysql_get_server_info` `(MYSQL *conn);`

  返回值：服务器的版本号。它被表示为一个以空结尾的字符串，如"5.0.60-debug- log"。这个版本号由版本序号和一个或者多个后缀组成。后缀的含义可以在附录 C 里对 `VERSION()` 函数的介绍里查到。

- `unsigned long`
  `mysql_get_server_version` `(void);`

  返回一个代表服务器版本的整数。该整数的格式为 *XYYZZ*，其含义是第 *X* 个系列里第 *YY* 次正式发布的第 *ZZ* 个版本。比如说，如果你使用的是 MySQL 5.1.25 版，这个函数将返回 `50125`。

- `const char *`
  `mysql_info` `(MYSQL *conn);`

  返回值：一个以空字符结尾的字符串。这个字符串对最近一次执行的以下几种语句的效果进行了描述。以下内容每两行为一组，第一行是语句，第二行则是相应的字符串格式：

```
ALTER TABLE ...
  Records: 0 Duplicates: 0 Warnings: 0
INSERT INTO ... SELECT ...
  Records: 0 Duplicates: 0 Warnings: 0
INSERT INTO ... VALUES (...),(...),...
  Records: 0 Duplicates: 0 Warnings: 0
LOAD DATA ...
  Records: 0 Deleted: 0 Skipped: 0 Warnings: 0
UPDATE ...
  Rows matched: 0 Changed: 0 Warnings: 0
```

  以上内容中的数字“0”将根据你具体执行的语句而变化。
  只有同时插入两个或两个以上记录的 `INSERT INTO...VALUES` 语句才会使 `mysql_info()`调用返回一个非 `NULL` 值。对于那些没有出现在上面的语句，`mysql_info()`的返回值将永远是 `NULL`。
  此外，`mysql_info()`所返回的字符串是用服务器当前使用的语言写出来的，所以你不能想当然地认为能够通过查找特定单词来解释它。

- `const char *`
  `mysql_stat` `(MYSQL *conn);`

  返回值：服务器的状态信息，它被表示为一个以空结尾的字符串。如果执行出错，则返回 `NULL` 值。这个字符串的格式并不固定，下面是它在最近一些 MySQL 版本里的样子：

```
Uptime: 2153150 Threads: 6 Questions: 1306220 Slow queries: 271 Opens: 1260
Flush tables: 1 Open tables: 64 Queries per second avg: 0.607
```

这个字符串的各组成部分的含义如下所示。

- Uptime。服务器运行时间，以秒为计算单位。
- Threads。服务器里现在正在运行的线程的个数。
- Questions。服务器已经执行过的语句个数。
- Slow queries。慢查询的个数。如果服务器用来处理某个语句的时间超出了服务器变量 long_query_time 所设定的时间，这个查询就将被认为是一个“慢查询”。
- Opens。服务器曾经打开过的数据表的个数。
- Flush tables。服务器曾经执行过的 FLUSH、REFRESH 和 RELOAD 语句的总个数。
- Open tables。服务器现在打开的数据表的个数。
- Queries per second。Questions 和 Uptime 的比值。

mysql_stat()函数所返回的信息与 mysqladmin status 命令的输出报告有着同样的内容(mysqladmin 程序正是通过调用这个函数来获取信息的)。

- unsigned long
  mysql_thread_id (MYSQL *conn);

  返回值：与当前连接相关联的服务器线程的 ID 号（与 CONNECTION_ID() SQL 函数返回的值相同)。你可以把这个 ID 号当做线程标识参数值传递给 KILL 语句。
  注意：应在你需要这个值时才调用 mysql_thread_id()函数。如果你获得了这个值并把它保存了起来，等你稍后再想使用这个值的时候，它有可能已经发生了改变。这种情况多发生在你掉线后又（比如说，使用 mysql_ping()函数）重新建立起来的场合；此时，服务器将给新连接重新分配一个线程标识符。
- unsigned int
  mysql_warning_count (MYSQL *conn);

  返回最近一条在执行时生成了警告消息的语句所生成的警告消息的个数。

G.3.7　事务控制例程

在这一节里，我们将对控制事务处理过程的函数进行介绍。

- my_bool
  mysql_autocommit (MYSQL *conn, my_bool mode);

  如果 mode 参数为真（非零），在当前连接里启用自动提交功能；否则，禁用该功能。如果调用成功，这个函数将返回零；否则，它将返回一个非零值。
- my_bool
  mysql_commit (MYSQL *conn);

  提交当前事务。如果调用成功，这个函数将返回零；否则，它将返回一个非零值。从 MySQL 5.0.3 版开始，这个函数会受到 completion_type 系统变量值的影响。
- my_bool
  mysql_rollback (MYSQL *conn);

回滚当前事务。如果调用成功，这个函数将返回零；否则，它将返回一个非零值。从 MySQL 5.0.3 版开始，这个函数会受到 completion_type 系统变量值的影响。

G.3.8　多结果集例程

本节里的例程只有在多语句执行功能已被启用的情况下才能使用。你可以通过在使用 mysql_real_connect()函数打开连接时给出 CLIENT_MULTI_STATEMENTS 标志来启用这个功能。还可以使用 mysql_set_server_option()函数为一个已经打开的连接启用多语句执行功能。

mysql_real_query()或 mysql_query()函数都可以把将要执行的语句发送到服务器。如果需要发送多条语句，必须把它们放在同一个字符串里发送，彼此之间要用分号隔开。

在 7.8 节可以找到一些演示如何使用这些例程的例子。

- my_bool
  mysql_more_results (MYSQL *conn);

  如果还有语句结果在等待读取，这个函数将返回一个非零值；否则，返回零。为了开始处理下一个结果，你必须调用 mysql_next_result()函数。
- int
  mysql_next_result (MYSQL *conn);

  如果还有语句结果在等待读取的话，开始处理下一个结果。在调用了这个函数之后，你就可以按照单独执行一条语句的流程处理结果了。
  如果还有更多的结果在等待处理，mysql_next_result()函数将返回 0；如果没有更多的结果，它将返回-1；如果发生错误，它将返回一个大于零的值。

G.3.9　预处理语句例程

本节里的例程在它们实现的二进制客户/服务器协议里提供了对预处理语句 API 的支持。它们可以划分为以下几个类别。

- 出错报告例程。用来获取出错代码和出错消息。
- 构造和执行例程。用来构造 SQL 语句和把它们发送到服务器。
- 结果集处理例程。用来对那些会返回一些数据的语句的执行结果进行处理。

在刚开始的时候，预处理语句机制只支持以下语句：CREATE TABLE、DELETE、DO、INSERT、REPLACE、SELECT、SET、UPDATE 语句和 SHOW 语句的绝大多数变体。这个语句清单在 MySQL 5.1 系列版本里有了显著的扩展，其最新内容请查阅《MySQL 参考手册》。

1. 预处理语句出错报告例程

本节将要介绍的例程使我们能够确定和报告预处理语句执行出错的原因。有可能出现的出错代码和出错消息开列在 errmsg.h、mysqld_error.h 和 sql_state.h 等 MySQL 头文件里。

- unsigned int
  mysql_stmt_errno (MYSQL_STMT *stmt);

  为最近调用的会返回一个状态值的预处理语句例程返回一个出错代码。如果没有发生任何错误，该出错代码将是零；否则，它将是一个非零值。

```
if (mysql_stmt_errno (stmt) == 0)
  printf ("Everything is okay\n");
```

```
else
  printf ("Something is wrong!\n");
```

- const char *

  mysql_stmt_error (MYSQL_STMT *stmt);

  为最近调用的会报告一个状态值的预处理语句例程返回一个以空字符为结束标记的出错消息字符串。如果没有发生任何错误，该返回值将是一个空白字符串（注意：是零长度的字符串""，不是一个 NULL 指针）。虽然 mysql_stmt_error() 函数通常是在知道发生错误以后才调用的，但它的返回值本身也可以用来检测是否发生了错误：

```
const char*err = mysql_stmt_error (stmt);
if (err[0] == '\0')                    /* empty string? */
  printf ("Everything is okay\n");
else
  printf ("Something is wrong!\n");
```

- const char *

  mysql_stmt_sqlstate (MYSQL_STMT *stmt);

  为最近调用的会返回一个状态值的预处理语句例程返回一个以空字符为结束标记的 SQLSTATE 出错代码字符串。这个代码是一个由 5 个字符构成的字符串。SQLSTATE 值取材于 ANSI SQL 和 ODBC 标准。代码"00000"的含义是“没有任何错误”。代码"HY000"的含义是“一般错误”，凡是没有专用 SQLSTATE 代码的 MySQL 错误都将用这个代码来代表。

```
if (strcmp (mysql_stmt_sqlstate (stmt), "00000") == 0)
  printf ("Everything is okay\n");
else
  printf ("Something is wrong!\n");
```

2. 预处理语句构造和执行例程

本节里的函数能够把预处理 SQL 语句发送到服务器。这些例程的语句字符串参数每次只能包含一条 SQL 语句，并且不允许在语末尾添加分号（“;”）或“\g”序列。在 SQL 语句末尾添加“;”字符或“\g”序列是 mysql 客户程序的要求，不是 C 客户库的。

你可以在 7.9 节里找到一些演示如何使用这些函数的例子。

- my_bool

  mysql_stmt_bind_param (MYSQL_STMT *stmt, MYSQL_BIND *bind_array);

  给定一个预处理语句的处理程序 stmt，mysql_stmt_bind_param() 函数将把一组数据值绑定到该语句中的“?”占位符。bind_array 是一个 MYSQL_BIND 结构数组的地址。预处理语句中的每一个占位符都必须在这个数组里有一个与之对应的结构。如果绑定操作成功，这个函数将返回零；否则，它将返回一个非零值。

- my_bool

  mysql_stmt_close (MYSQL_STMT *stmt);

  关闭给定的预处理语句处理程序并释放与之相关联的所有资源，该处理程序上还没来得及处理的结果将被丢弃。如果调用成功，这个函数将返回零；否则，它将返回一个非零值。

  在关闭某个语句处理程序之后，千万不要再使用它去进行任何操作。

  如果某个客户在断开与服务器的连接时还有与之相关联的预处理语句留在服务器上，服务器将丢弃那些语句。

- ❑ MYSQL_STMT *
  mysql_stmt_init (MYSQL *conn);

  构造并初始化一个 MYSQL_STMT 处理程序。如果调用成功，这个函数将返回一个指向该处理程序的指针；如果无法构造该处理程序，它将返回 NULL。
  在用完某个处理程序后，千万不要忘记使用 mysql_stmt_close()函数把它释放掉。
- ❑ int
  mysql_stmt_execute (MYSQL_STMT *stmt);

  执行与给定语句处理程序相关联的预处理语句。如果该语句执行成功，这个函数将返回零；否则，它将返回一个非零值。
  在执行一条语句之前，如果该语句包含有“?”占位符，你必须通过调用 mysql_stmt_bind_param()函数为每个占位符绑定一个数据值。
  在成功地执行一条语句之后，如何对其执行结果进行处理要根据它是否会生成一个结果集来决定。对于不会生成任何结果集的语句，可以调用 mysql_stmt_affected_rows()函数以确定它实际插入、删除或刷新了多少个数据行。对于会生成结果集的语句，其结果集的元数据可以用 mysql_stmt_result_metadata()函数来检索。取回结果的基本套路是：先用 mysql_stmt_bind_result()函数把结果集里的数据列绑定到相应的缓存变量，再用 mysql_stmt_fetch()函数检索数据行，最后用 mysql_stmt_free_result()函数释放该结果集。
- ❑ int
  mysql_stmt_prepare (MYSQL_STMT *stmt,
  const char *stmt_str,
  unsigned long length);

  给定一个计数字符串形式的 SQL 语句，这个函数将把该语句发送到服务器接受预处理，并把语句处理程序 stmt 和该预处理语句关联在一起。该语句的文本由 stmt_str 参数给出，它的长度由 length 设定。这种字符串允许包含（包括空字节在内的）二进制数据。
  如果调用成功，这个函数将返回零；如果失败，它将返回一个非零值。
  在 stmt_str 参数所给出的语句里，允许使用“?”字符作为占位符；等以后执行这条语句时，必须把一些数据值绑定到这条语句以替换其中的占位符。
- ❑ my_bool
  mysql_stmt_reset (MYSQL_STMT *stmt);

  对给定的预处理语句处理程序进行重置，让它恢复到刚调用完 mysql_stmt-prepare()函数时的状态。
- ❑ MYSQL_RES *
  mysql_stmt_result_metadata (MYSQL_STMT *stmt);

  在成功地调用 mysql_stmt_execute()函数执行了一条语句之后，如果该语句是一条会生成结果集的语句，你就可以调用 mysql_stmt_result_metadata()函数去获得它的结果集元数据。返回值是一个指向一个 MYSQL_RES 结构的指针。除了不包含任何结果数据，调用这个函数获得的“结果集”和你为同一条语句的非预处理版本调用 mysql_store_result()函数而获得的结果集完全一样。只需把它返回的结构指针传递给 mysql_fetch_field()、mysql_fetch_

fields()和 mysql_num_fields()等接受 MYSQL_RES 参数的函数，就可以获得关于结果数据列的相应信息。在用完这个结构之后，别忘了把它传递到 mysql_free_result()函数以释放它。对于那些不会生成一个结果集的预处理语句，这个函数将返回 NULL 以表明“该语句没有元数据信息”。

- my_bool
  mysql_stmt_send_long_data (MYSQL_STMT *stmt,
  ```
                              unsigned int param_num,
                              const char *data,
                              unsigned long length);
  ```
  这个函数用来向预处理语句中的给定参数传递一个超长的 BLOB 或 TEXT 值。param_num 是将接受被传递数据的参数的编号，其取值范围是从 0 到 mysql_stmt_param_count(stmt)−1。将被传递的数据存放在 data 指针所指向的那个缓冲区里，length 用来设定将要传递多少个字节。

3. 预处理语句结果集处理例程

在执行一条会生成结果集的预处理语句时，本节里的函数能够检索结果集和访问它的内容。

你可以在 7.9 节里找到一些演示如何使用这些函数的例子。

- my_ulonglong
  mysql_stmt_affected_rows (MYSQL_STMT *stmt);

  这个函数是 mysql_affected_row()函数在预处理语句机制中的实现，它应该在 mysql_stmt_execute()函数返回后调用。对于不会返回任何结果集的预处理语句，这个函数将返回该语句实际插入、删除或刷新的数据行的个数。对于会返回一个结果集的预处理语句，这个函数的行为和 mysql_num_rows()函数相同。

  这个函数将返回一个 my_ulonglong 值。关于如何打印这种类型的值的问题，请参阅 G.2.1 节。

- my_bool
  mysql_stmt_attr_get (MYSQL_STMT *stmt,
  ```
                      enum enum_stmt_attr_type attr_type,
                      void *attr);
  ```
  读取给定预处理语句处理程序的给定属性。attr_type 参数的可取值见 mysql_stmt_attr_set()函数条目里的描述。attr 参数是一个指针，这个函数将把读取到的属性值写入该指针所指向的变量。（特例：在 MySQL 5.1.7 之前，在读取 STMT_ATTR_UPDATE_MAX_LENGTH 属性时，应该向 attr 参数传递一个指向 unsigned int 变量而不是一个指向 my_bool 变量的指针。）

  ```
  my_bool attr;
  if (mysql_stmt_attr_get (stmt, STMT_ATTR_UPDATE_MAX_LENGTH, &attr) == 0)
    printf ("Attribute gotten successfully\n");
  else
    printf ("Could not get attribute\n");
  ```
  如果成功地读取到了给定属性，这个函数将返回零；如果该属性的类型未知，则返回一个非零值。

- my_bool
  mysql_stmt_attr_set (MYSQL_STMT *stmt,
  ```
                      enum enum_stmt_attr_type attr_type,
                      const void *attr);
  ```

设置给定预处理语句处理程序的给定属性。attr_type 用来给出将被设置的属性，该属性的新设置值存放在 attr 所指向的变量里。

attr_type 参数的可取值有以下几个。

- STMT_ATTR_UPDATE_MAX_LENGTH。这个属性控制着 mysql_stmt_store_result()函数是否要为结果集数据列计算 max_length 元数据值。启用或禁用这个属性的办法是让 attr 参数指向一个被设置为“真”或“假”的 my_bool 值。max_length 是默认禁用的
- STMT_ATTR_CURSOR_TYPE。这个属性控制着在调用 mysql_stmt_execute()函数执行一条预处理语句时应该为该语句选用什么类型的游标。此时应该让 attr 参数指向一个被设置为 CURSOR_TYPE_NO_CURSOR（默认设置）或 CURSOR_TYPE_READ_ONLY 的 unsigned long 值。
- STMT_ATTR_PREFETCH_ROWS。这个属性控制着在使用游标的情况下每次应该从服务器取回多少个数据行。此时应该让 attr 参数指向一个代表着数据行个数的 unsigned long 值；这个值应该大于或等于 1（默认设置是 1）。

下面这个例子演示了如何启用为结果集计算 max_length 元数据值的属性：

```
my_bool attr = 1;
if (mysql_stmt_attr_set (stmt, STMT_ATTR_UPDATE_MAX_LENGTH, &attr) == 0)
  printf ("Attribute set successfully\n");
else
  printf ("Could not set attribute\n");
```

如果属性设置成功，这个函数将返回零；如果该属性的类型未知，则返回一个非零值。

❑ my_bool

mysql_stmt_bind_result (MYSQL_STMT *stmt, MYSQL_BIND *bind_array);

给定一个预处理语句处理程序 stmt，这个函数将为之提供一个 MYSQL_BIND 结构数组用于取回结果集里的数据行。bind_array 是一个以 MYSQL_BIND 结构为元素的数组的地址，结果集里的每一个数据列都必须在该数组里有一个与之对应的结构，每当你调用 mysql_stmt_fetch()函数取回一个结果数据行的时候，该数据行各数据列的值就将被返回到这些 MYSQL_BIND 结构里。如果绑定操作成功，这个函数将返回零；否则，它将返回一个非零值。

MYSQL_BIND 结构和结果集数据列之间的绑定操作必须在开始检索数据行之前完成，而这些结构所指向的缓冲区必须大到足以容纳将被取回的数据值。在检索一个结果集的过程中，你还可以调用 mysql_stmt_bind_result()函数把结果集数据列绑定到另外一组 MYSQL_BIND 结构，而 mysql_stmt_fetct()函数将使用你最近一次绑定的结构数组来取回数据。

❑ void

mysql_stmt_data_seek (MYSQL_STMT *stmt, my_ulonglong row_num);

跳转到结果集里的指定数据行。row_num 参数的可取值范围是从 1 到 mysql_stmt_num_rows (stmt) −1。如果 row_num 值超出范围，后果将无法预料。

mysql_stmt_data_seek()函数要求整个结果集都已经被取回到客户端内存里，所以你在执行了一条预处理语句后必须先调用过 mysql_stmt_store_result()函数才能使用它。

千万不要把 mysql_stmt_data_seek()函数与 mysql_stmt_row_seek()函数弄混了，后者需要你提供一个数据行偏移值——就像 mysql_stmt_row_tell()函数的返回值那样——而不是一个数据行编号。

❑ unsigned int
mysql_stmt_field_count (MYSQL_STMT *stmt);

你在调用 mysql_stmt_prepar()函数获得一个预处理语句处理程序后随时都可以调用它。如果给定语句处理程序所代表的语句在执行时将生成一个结果集,这个函数将返回该结果集里的数据列的个数;如果那条语句不会生成一个结果集(比如说,如果那是一条 INSERT 或 UPDATE 语句),这个函数将返回零。

❑ int
mysql_stmt_fetch (MYSQL_STMT *stmt);

在成功地调用 mysql_stmt_execute()函数执行了一条会生成一个结果集的预处理语句之后,无论是否调用过 mysql_stmt_store_result()函数把那个结果集取回到客户端内存,你都可以调用 mysql_stmt_fetch()函数去检索那个结果集里的数据行。用来取回结果数据列的缓冲区必须已经通过 mysql_stmt_bind_result()函数调用绑定到了 MYSQL_BIND 结构。

如果成功地取回了一个数据行,mysql_stmt_fetch()函数将返回零;如果没有数据行可供取回,它将返回 MYSQL_NO_DATA;如果发生错误,它将返回 1。在成功地取回了一个数据行之后,你就可以通过与结果集相绑定的那些 MYSQL_BIND 结构去访问该数据行各数据列的值了。

❑ int
```
mysql_stmt_fetch_column (MYSQL_STMT *stmt,
                         MYSQL_BIND *bind,
                         unsigned int col_num,
                         unsigned long offset);
```

这个函数用来从当前结果集数据行的一个给定数据列提取数据。如果提取成功,返回零;如果发生错误,返回一个非零值。bind 是一个 MYSQL_BIND 结构,而你已经对它进行了必要的设置以表明打算提取的数据值是什么类型、把它提取到哪个缓冲区、需要提取多少个字节等。col_num 是你打算提取的那个数据列的编号,它的取值范围是从 0 到 mysql_stmt_field_count(stmt)−1。offset 是对数据列值进行提取时的起始位置偏移值,0 表示从该值的开头开始。

❑ my_bool
mysql_stmt_free_result (MYSQL_STMT *stmt);

释放与给定预处理语句处理程序相关联的结果集所占用的内存。如果释放成功,返回零;否则,返回一个非零值。未被取回的数据行都将被丢弃。为了尽可能地避免造成内存泄漏,应该遵循这样一个原则:只要不再需要从结果集取回数据行,就应该立刻调用 mysql_stmt_free_result()函数释放它。这一原则在你有可能使用同一个处理程序生成多个结果集的时候更要牢记。

❑ my_ulonglong
mysql_stmt_insert_id (MYSQL_STMT *stmt);

这个函数是 mysql_insert_id()函数在预处理语句机制中的实现。它应该在你调用 mysql_stmt_execute()函数执行完一条会生成一个 AUTO_INCREMENT 值的语句之后使用。

这个函数将返回一个 my_ulonglong 值。关于如何打印这种类型的值的问题,请参阅 G.2.1 节。

❑ my_ulonglong
mysql_stmt_num_rows (MYSQL_STMT *stmt);

返回结果集里的数据行的个数——如果你已经通过调用mysql_stmt_store_result()函数把结果集取回到客户端内存的话。如果你还没有调用过 mysql_stmt_store_ result()函数，mysql_stmt_num_rows()函数将返回零。

这个函数将返回一个 my_ulonglong 值。关于如何打印这种类型的值的问题，请参阅 G.2.1 节。

- int

  mysql_stmt_store_result (MYSQL_STMT *stmt);

  在默认的情况下，服务器在执行完一条预处理语句后，并不会把所生成的结果集发送给客户，而你每调用一次 mysql_stmt_fetch()函数只能从服务器取回一个数据行。如果在语句执行完毕之后、在开始取回结果集之前发出一个 mysql_stmt_store_result()函数调用，就可以把结果集完整地取回并缓存到客户端内存里，并让随后的 mysql_stmt_fetch()函数调用从结果集返回数据行。调用 mysql_stmt_store_result()函数还可以让结果集变成“可跳转的”，意思是你可以借助于 mysql_stmt_data_seek()、mysql_stmt_row_seek()和 mysql_stmt_ row_tell()函数在结果集里快速跳转到特定的位置。这几个函数只能在结果集被取回并缓存到客户端内存里以后才能使用，它们通过在结果集里移动一个数据行游标来实现其操作。

  出于性能方面的考虑，服务器在默认的情况下不会为结果集里的每个数据列计算其元数据当中的 max_lenght 值。如果想让服务器在你调用 mysql_stmt_store_result()函数时把这个值计算出来，提前用 mysql_stmt_set_attr()函数启用语句处理程序的 STMT_ATTR_UPDATE_MAX_LENGTH-FLAG 属性即可。

  其实你用不着调用 mysql_stmt_store_result()函数也可以通过调用 mysql_stmt_fetch()函数取回结果集里的数据行，但那将使你只能一个一个地从服务器取回数据行。

  在执行了一条不会生成任何结果集的语句后，调用 mysql_stmt_store_result()函数将不会有任何效果。

- unsigned long

  mysql_stmt_param_count (MYSQL_STMT *stmt);

  在成功地调用 mysql_stmt_prepare()函数对一条语句进行了预处理之后，mysql_stmt_param_count()函数调用将返回该语句里的参数（以“?”占位符表示）的个数。如果该语句不包含任何占位符，这个函数的返回值将是零。

- MYSQL_ROW_OFFSET

  mysql_stmt_row_seek (MYSQL_STMT *stmt, MYSQL_ROW_OFFSET offset);

  快速跳转到结果集里的特定数据行。这个函数的行为和 mysql_stmt_data_seek()函数很相似，但 offset 值并不是一个数据行序号。offset 是一个必须通过调用 mysql_stmt_row_tell()或 mysql_stmt_row_seek()函数获得的 MYSQL_ROW_OFFSET 值，或者是零；offset 取值为零的含义是跳转到结果集里的第一个数据行。

  mysql_stmt_row_seek()函数的返回值是前一次跳转使用的偏移值。

  mysql_stmt_row_seek()函数要求整个结果集已经被取回到了客户端内存，你只能在语句执行完毕后已经调用过 mysql_stmt_store_result()函数的情况下才能使用它。

- MYSQL_ROW_OFFSET

  mysql_stmt_row_tell (MYSQL_STMT *stmt);

  返回一个代表当前数据行在结果集里的位置的偏移值。这个偏移值并不一个数据行序号，你只能

把它传递给 mysql_stmt_row_seek()函数，不能把它传递给 mysql_stmt_data_seek()函数。mysql_stmt_row_tell()函数要求整个结果集已经被取回到了客户端内存，你只能在语句执行完毕后已经调用过 mysql_stmt_store_result()函数的情况下才能使用它。

G.3.10 系统管理例程

本节里的函数能够对服务器的操作状态进行调控。

- int
  mysql_refresh (MYSQL *conn, unsigned int options);

  这个函数类似于 SQL 语言中的 FLUSH 和 RESET 语句，但它可以让服务器一次“清除”好几种东西。如果调用成功，这个函数将返回零；如果调用失败，它将返回一个非零值。

  options 值应该由一个或多个下列设置项构成。你必须具备 RELOAD 权限才能进行这些操作。

  - REFRESH_GRANT
    重新加载权限数据表的内容，其效果相当于执行了一条 FLUSH PRIVILEGES 语句。
  - REFRESH_HOSTS
    “清除”主机缓存，其效果相当于执行了一条 FLUSH HOSTS 语句。
  - REFRESH_LOG
    通过先关闭再打开日志文件的办法“清除”它们，服务器已经打开的日志都将受其影响；效果相当于执行了一条 FLUSH LOGS 语句。
  - REFRESH_MASTER
    让复制主服务器删除在二进制日志索引文件里列出的二进制日志文件并截短索引文件，其效果相当于执行了一条 RESET MASTER 语句。
  - REFRESH_SLAVE
    让复制从服务器“忘记”它在主日志里的位置，其效果相当于执行了一条 RESET SLAVE 语句。
  - REFRESH_STATUS
    把状态变量重新初始化为零值，其效果相当于执行了一条 FLUSH STATUS 语句。
  - REFRESH_TABLES
    关闭所有打开的数据表，其效果相当于执行了一条 FLUSH TABLES 语句。
  - REFRESH_THREDS
    “清除”线程缓存，这个操作没有与之相当的 SQL 语句。

  上面列出的这些设置项都是些二进制位值，所以你可以用“|”或“+”操作符把它们叠加在一起使用。比如说，下面两个表达式是等效的：

```
REFRESH_LOG | REFRESH_TABLES
REFRESH_LOG + REFRESH_TABLES
```

- int
  mysql_set_server_option (MYSQL *conn,
  enum enum_mysql_set_option option);

  设置一个服务器选项。如果该选项设置成功，返回零；否则，返回一个非零值。现阶段允许设置的选项只有 MYSQL_OPTION_MULTI_STATEMENTS_ON 和 MYSQL_OPTION_MULTI_STATEMENTS_OFF 两种，它们分别用来启用或禁用服务器的多语句执行能力。

  用 MYSQL_OPTION_MULTI_STATEMENTS_ON 选项启用多语句执行能力不会同时启用多结果集。

这和通过 mysql_real_connect()函数设置 CLIENT_MULTI_STATEMENTS 选项会同时启用 CLIENT_MULTI_RESULTS 选项的情况是不同的。

- int

  mysql_shutdown (MYSQL *conn, enum mysql_enum_shutdown_level level);

  通知服务器关机。你必须具备 SHUTDOWN 权限才能这么做。第二个参数的值在现阶段只能是 SHUTDOWN_DEFAULT，其他的关机级别还有待实现。

  如果调用成功，这个函数将返回零；如果调用失败，它将返回一个非零值。

G.3.11　线程客户例程

本节里的函数是用来编写多线程客户程序的。

- void

  mysql_thread_end (void);

  释放由 mysql_thread_init()函数初始化的所有与线程有关的变量。为避免造成内存泄漏，你应该明确地调用这个函数，结束你创建的每一个线程。

- my_bool

  mysql_thread_init (void);

  初始化与线程有关的变量。你应该为你创建的每一个将会调用 MySQL 函数的线程调用这个函数。在此基础上，你还应该在结束每一个这样的线程之前调用 mysql_thread_end()函数。

- unsigned int

  mysql_thread_safe (void);

  如果客户库支持线程，返回 1；否则，返回 0。这个函数的返回值反映了这个 MySQL 服务器在编译时是否启用了--enable_thread_safe_client 选项。

G.3.12　调试例程

这些函数使你能够在当前连接的客户端或服务器端生成调试信息。这需要 MySQL 软件已经通过编译而具备了调试支持能力(在配置 MySQ 发行版本时使用--with-debug 或--with-debug=full 选项。后者可以让你获得更详细的调试信息并启用 safemalloc 库，该库在分配内存时会进行更严格的检查)。

- void

  mysql_debug (const char *debug_str);

  用字符串 debug_str 进行一次 DEBUG_PUSH 操作，该字符串的格式在《MySQL 参考手册》里有详细的描述。

  为了使用 mysql_debug()函数，你使用的客户库必须已经通过编译而具备了调试支持能力。

- int

  mysql_dump_debug_info (MYSQL *conn);

  让服务器把调试信息写入日志。你必须具备 SUPER 权限才能这么做。

  如果调用成功，这个函数将返回零；如果调用失败，它将返回一个非零值。

附录 H

Perl DBI API 指南

本附录对 Perl DBI 应用程序编程接口进行了介绍。利用这个 API 所提供的方法和属性，Perl 脚本将能与数据库服务器通信并访问数据库。本附录还将介绍 MySQL 数据库驱动模块 DBD::mysql 向 DBI 提供的 MySQL 扩展功能。

DBI 现阶段的最新版本是 1.601，但这个附录里的大部分内容同样适用于早期版本。为稳妥起见，我把最低要求设定为 DBI 1.40 版，而这要求 Perl 至少是 5.6.0 版（5.6.1 版比 5.6.0 版更好）。我为 DBD::mysql 设定的最低要求是 3.0000_0 版。DBI 和 DBD::mysql 在上述版本之后引入的新功能将在讨论中随时注明。如果你的程序需要使用 `mysql_server_prepare` 支持，建议使用 DBD::mysql 3.0009 或更高的版本，因为它们对这个功能在早期版本里的默认设置又做了一些改进。

运行以下程序可以确定 DBI 和 DBD::mysql 的版本：

```
#!/usr/bin/perl
# dbi-version.pl - display DBI and DBD::mysql versions
use strict;
use warnings;
use DBI;
print "DBI::VERSION: $DBI::VERSION\n";
use DBD::mysql;
print "DBD::mysql::VERSION: $DBD::mysql::VERSION\n";
```

有些 DBI 模块方法和属性没有在本附录里介绍，它们或者不适用于 MySQL，或者是一些比较新的实验性方法，还有待于完善或者最终被放弃。有些与 MySQL 有关的 DBD 方法也没有收录在此附录里，因为它们已经过时了。如果你想了解这些新、旧功能，请自行查阅 DBI 文档或者 MySQL DBD 文档，可以用下面这些命令来查看：

```
% perldoc DBI
% perldoc DBI::FAQ
% perldoc DBD::mysql
```

相关信息可参见 http://dbi.perl.org/。

这里收录的都是一些代码片段。第 8 章有一些完整的客户脚本以及编写它们的指导意见。

H.1 编写脚本

如果想在 Perl 脚本里使用 DBI 模块，就必须把下面这行代码加到你的脚本代码里去：

```
use DBI;
```

但你不必用一条 `use` 指令来导入某个具体的 DBD 级别的模块，因为 DBI 会在你连接服务器时自

动启用正确的模块。

在正常情况下，DBI 脚本需要使用 connect()方法来打开一个与 MySQL 服务器的连接，再使用 disconnect()方法来关闭这个连接。只有在这个连接处于打开状态，你才可以执行 SQL 语句。能够用来执行查询命令的方法有好几种，取决于语句的类型。非 SELECT 查询通常是用 do()方法来直接执行的。SELECT 查询通常要分成以下几个步骤来执行：先把查询命令传递到 prepare()方法，再调用 execute()方法去执行它，最后用一个数据行取回方法（比如 fetchrow_array()或 fetchrow_hashref()）在一个循环语句里以一次一个数据行的方式来取回并处理它的查询结果。

当你通过 DBI 脚本来执行语句时，每个语句字符串只能包含一条 SQL 语句，并且不允许在末尾加上分号（;）或“\g”结束符；结束符“;”和“\g”都只是 mysql 客户程序使用的记号，不用于 DBI 脚本。

H.2 DBI 方法

在附录 G 和附录 I 里，我们在介绍 C 语言函数和 PHP 函数时给出的是一个完整的函数原型，那些函数的返回值类型和输入参数类型都有明确的定义。但在这篇附录里，我们采用了一种不同的格式。我们将用一些变量来代表输入参数和返回值，而它们的类型将通过变量名的第一个字符来隐含地表明：'$'代表一个标量值，'@'代表一个数组，'%'代表一个散列值（即关联数组）。此外，如果变量名的前面还有一个'\'字符，则表示它是一个指向该变量的引用指针而不是该变量本身。如果变量名的前面还有一个 ref 后缀，则表示该变量的值是一个引用指针。

表 H-1 列出了在这篇附录里比较常见的一些变量和它们的基本含义。

表H-1 常用的Perl DBI变量名

变 量 名	含 义
$drh	一个句柄，指向一个驱动模块对象
$dbh	一个句柄，指向一个数据库对象
$sth	一个句柄，指向一条语句（查询）对象
$fh	一个句柄，指向一个已被打开的文件
$h	一个句柄，其具体含义需要根据上下文才能确定
$rc	返回代码，来自那些会返回一个真值或假值的操作
$rv	返回值，来自那些会返回一个整数的操作
$rows	返回值，来自那些会返回一个数据行计数值的操作
$str	返回值，来自那些会返回一个字符串的操作
@ary	一个数组，表示一组值
@row_ary	一个数组，存放着从一条查询命令的结果集里取回的一个数据行

此外，很多 DBI 方法都接受一个散列参数%attr，这个参数给出的属性值将影响到这个方法的具体工作情况。在用到这个散列参数的时候，你应该把它的引用指针传递给有关方法，其具体做法有两种。

- 先设置好散列值%attr 的内容，然后调用有关方法，并把这个散列值的引用指针用作那个方法的一个输入参数，如下所示：

```
my %attr = (AttrName1 => value1, AttrName2 => value2);
$ret_val = $h->method (..., \%attr);
```

- 直接在方法调用中使用一个没有名字的散列值，如下所示：

```
$ret_val = $h->method (..., {AttrName1 => value1, AttrName2 => value2});
```

方法或函数的调用方式将由调用序列表明。“DBI->”表明那是一个 DBI 类的方法，“DBI::”表明那是一个 DBI 函数，“$DBI::”表明那是一个 DBI 变量。对于那些需要通过句柄来调用的方法，句柄的名称将表明该方法的作用范围。“$dbh->”表明那是一个数据库句柄方法，“$sth->”表明那是一个语句句柄方法，“$h->”表明那个方法可以用各种不同的句柄来调用。对于那些允许省略的可选信息，我们将把它们放在一对方括号（[]）里。请看下面这个调用序列示例：

```
@row_ary = $dbh->selectrow_array ($statement, [\%attr [, @bind_values]]);
```

上面这个调用序列里的“$dbh->”表明 selectrow_array()方法是作为一个数据库句柄方法调用的。这个方法的输入参数是$statement（一个标量值）、%attr（一个散列值，因为它前面有一个“\”字符的缘故，所以我们知道它必须被传递为一个引用）和@bina_values（一个数组），而且第 2 和第 3 个输入参数是可选的。它的返回值@row_ary 是一个数组，里面存放着这个方法所返回的那个数据行的内容。

在介绍各方法时，我们会给出它在执行出错时的返回值，但这个值只有禁用了 RaiseError 属性的情况下才会被返回。如果 RaiseError 属性处于启用状态，执行出错的方法将不会返回而是会抛出一个例外并导致异常自动退出执行。

在下面的内容里，“SELECT 查询”代表着能够返回一些数据行的 SELECT 语句，比如 SELECT、DESCRIBE、EXPLAIN 或 SHOW。

H.2.1 DBI类方法

在这一节里，各方法的%attr 参数可以用来设定该方法的处理属性。(如果某个属性参数被省略或者取值为 undef，则表示“不具备该属性”。）对与 MySQL 有关的 DBI 方法来说，最重要的属性是 PrintError、RaiseError 和 AutoCommit。被传递给 connect()或 connect_cached()方法的属性将成为这两个方法所返回的结果数据库句柄的一部分。比如说，下面这行代码在创建句柄时启用了 RaiseError 属性，这就激活了脚本的出错自动退出机制：如果脚本在调用与句柄相关联的任何一个方法时发生了一个 DBI 错误，脚本就将自动退出执行：

```
$dbh = DBI->connect ($data_source, $user_name, $password, {RaiseError => 1});
```

我们将在 H.4 节里对 PrintError、RaiseError 和 AutoCommit 属性做详细的讨论。

- **@ary = DBI->available_drivers ([$quiet]);**

  返回一份可用 DBI 模块的名单。可选参数$quiet 的默认值是 0，意思是在发现有多个驱动模块有同样的名字时报告一条警告消息。如果不想看到这样的警告消息，把$quiet 的值设置为 1 即可。

- ```
  $dbh = DBI->connect    ($data_source,
                          $user_name,
                          $password
                          [, \%attr]);
  ```

  建立与服务器的连接。如果调用成功，它将返回一个数据库句柄，如果调用失败，则返回 undef。在成功地建立起一个连接之后，如果你想关闭它，就需要使用 connect()所返回的数据库句柄来调用 disconnect()方法。如下所示：
  ```

```
$dbh = DBI->connect ("DBI:mysql:sampdb:localhost",
                     "sampadm", "secret", \%attr)
         or die "Could not connect\n";
$dbh->disconnect ();
```

数据源允许以几种不同的格式给出。数据源必须以“DBI:mysql:”开头，其中，“DBI”这几个字符的大小写可以随意，“mysql”这几个字符却必须是小写。注意，因为第二个冒号（这个冒号不允许省略）后面的全部内容都将由具体的数据库驱动模块负责解释，所以本附录后面内容里介绍的语法不一定适用于 DBD::mysql 以外的其他数据库驱动模块。

在第二个冒号的后面，你还可以给出一个数据库名和一个主机名，放在数据源字符串的开头部分。如下所示：

```
$data_source = "DBI:mysql:db_name";
$data_source = "DBI:mysql:db_name:host_name";
```

可用 *db_name* 或 *database_name* 指定数据库。主机名可用 *host_name* 或 *host=host_name* 指定。

还可以在%attr 参数里传递 Username 和 Password 属性以设定用户名和口令，这两个属性的优先级高于通过$user_name 和$password 参数传递的值。

```
my %attr = (Username => "sampadm", Password => "secret");
$dbh = DBI->connect ("DBI:mysql:sampdb:localhost",
                     "someuser", "somepass", \%attr)
         or die "Could not connect\n";
```

属性还可以在驱动程序名后面的数据源字符串里给出，以逗号为分隔符并用括号括起来。以这种方式设定的属性，其优先级高于通过%attr、$user_name 和$password 参数传递的属性值。

```
my $dsn = "DBI:mysql(Username=>sampadm,Password=>secret):sampdb:localhost";
$dbh = DBI->connect ($dsn, "someuser", "somepass", \%attr)
         or die "Could not connect\n";
```

你还可以在数据源字符串开头部分的后面紧接着以 *attribute* = *value* 的格式给出一些选项。这些选项必须以冒号加以分隔（即每个选项的前面都必须有一个冒号）。如下所示：

```
DBI:mysql:sampdb:localhost;mysql_socket=/tmp/mysql.sock;mysql_compression=1
```

MySQL 驱动模块能够识别并支持使用以下几个选项：

- host = *host_name*

  脚本将去连接的主机。对于 TCP/IP 连接，你还可以通过 *host_name:port_name* 格式或直接使用 port 属性来指定一个端口号。

  在 Unix 系统上，对本地主机 localhost 的连接将默认地使用一个 Unix 套接字来建立。（这需要你通过 mysql_socket 属性来给出一个套接字名。）如果想使用 TCP/IP 来连接本地主机，就需要以 host = 127.0.0.1 的形式来加以指定。

  在 Windows 系统上，以“.”为主机名的连接将使用一个命名管道（如果命名管道不可用，则使用 TCP/IP）连接至本地主机。在这种情况下，可能要使用 mysql_socket 选项来指定一个命名管道。

- port = *port_num*

  服务器主机的端口号。如果那是一条非 TCP/IP 连接（比如说，在 Unix 系统上使用 localhost 主机名建立的连接），这个选项将被忽略。

- mysql_client_found_rows = *val*
  返回的计数值的类型。对于 UPDATE 语句，MySQL 服务器可以返回它影响（改变）的数据行的个数或是它匹配到的（不管它们是否会被修改）数据行个数。比如说，如果某个 UPDATE 语句用 WHERE 子句选取了一个数据行，但只是把该数据行的值再次设置为它们的当前值，它匹配到的数据行个数将是 1，而它修改的数据行个数将是 0。把 mysql_client_found_rows 选项设置为 0 将禁用该选项，服务器将返回被修改的数据行个数；把这个选项设置为 1 将启用该选项，此时返回的是被匹配到的数据行个数。
  在默认的情况下，这个选项在 DBD::mysql 里是启用的。C 客户库的默认设置是返回被修改的数据行个数，与 DBD::mysql 截然相反。
- mysql_compression = 1
  要求客户与 MySQL 服务器之间的通信协议对传输数据进行压缩——如果双方都支持数据压缩的话。
- mysql_connect_timeout = *seconds*
  如果等待了 *seconds* 秒还没能建立起连接的话，向用户报告本次连接尝试失败。
- mysql_local_infile = *val*
  这个选项控制着 LOAD DATA 语句是否具备 LOCAL 能力。如果 MySQL 客户库里的默认设置是禁用 LOCAL 能力，把这个选项设置为 1 将启用之(但前提是服务器没有禁用 LOCAL 能力)。如果 LOCAL 能力在客户库里已被启用，把这个选项设置为 0 将禁用之。
- mysql_read_default_file = *file_name*
  在默认的情况下，DBI 脚本不会到任何 MySQL 选项文件里去寻找连接参数。这个选项使你能够强制它们去读取一个选项文件。*file_name* 值应该是一个完整的路径名。（否则，它将以当前子目录为起点进行解析，而你在不同的子目录里运行同一个脚本将得到不一样的结果。）
  在 Unix 系统上，如果你想让期某个脚本由多个用户使用，并想使用各自选项文件里指定的参数去连接服务器的话（把连接参数硬编码在脚本里让大家共享显然不是个好主意），你只需要把文件名设置为$ENV{HOME}/.my.cnf 就可以了。此后，当某个用户来运行这个脚本时，它就会去读取该用户的登录子目录里的.my.cnf 文件。
  在 Windows 系统上，你不能把一个带有硬盘盘符的选项文件名直接用作这个选项的设置值，这是因为紧跟在盘符后面的冒号字符（":"）还被 DBI 当做数据源字符串内部的一个分隔符来使用着。8.2.9 节给出了这个问题的解决方案。
- mysql_read_default_group = *group_name*
  告诉脚本应该到选项文件中的哪个选项组里寻找连接参数。如果只给出了 mysql_read_default_group 选项但没有给出 mysql_read_default_file 选项，脚本将读取标准选项文件。如果同时给出了上述两个选项，脚本将只读取后者指定的文件。
  [client]选项文件组是每个脚本必读的。利用 mysql_read_default_group 选项能够在[client]选项组的基础上再指定一个选项组让脚本读取。比如说，mysql_read_default_group = dbi 表明应该读取[client]和[dbi]两个选项组。如果想让脚本只读取[client]选项组，应该写成 mysql_read_default_group=client。
- mysql_server_prepare = *val*
  把这个选项设置为 1 将启用服务器端的语句预处理功能。如果把它设置为 0（默认设置），你在使用预处理语句时的所见所得其实是在客户端“模拟”出来的。

关于使用 mysql_server_prepare 选项所需要的 DBD::mysql 版本的信息见本附录开头的概述部分。

- mysql_socket = *socket_name*

  在 Unix 系统上，这个选项用来设定以 localhost 为主机名连接服务器时使用的 Unix 套接字文件的路径名。在 Windows 系统上，这个选项用来给出一个命名管道的名字。基于 TCP/IP 的连接将忽略这个选项。（在 Unix 系统上，没有使用 localhost 作为主机名的连接都是 TCP/IP 连接。）

- mysql_ssl=val

  mysql_ssl_ca_file=*file_name*
  mysql_ssl_ca_path=*dir_name*
  mysql_ssl_cipher=*str*
  mysql_ssl_client_cert=*file_name*
  mysql_ssl_client_key=*file_name*

  这些选项用于与服务器建立基于 SSL 的加密连接。把 mysql_ssl 选项设置为 0 将禁用 SSL。这个选项未指定或者是被设置为 1 的时候才允许建立 SSL 连接。其他选项用来设置 SSL 连接的各项属性，它们的含义和 C API 中的 mysql_ssl_set()函数的相应参数是一样的，详见附录 G 里关于这个函数的条目。如果启用了 mysql_ssl 选项，你最少也要设置 mysql_ssl_ca_file、mysql_ssl_client_cert 和 mysql_ssl_client_key 选项才能用上 SSL。使用这些选项的前提条件有两个，其一是链接在 DBD::mysql 里的 MySQL C 客户库必须支持 SSL；其二是你准备连接的那个服务器必须支持 SSL 连接。

- mysql_use_result=*val*

  这个选项影响着对结果集的检索行为。如果把这个选项设置为 0（默认设置），DBD::mysql 将使用 C API 中的 mysql_store_result()函数来检索数据行。如果把它设置为 1，DBD::mysql 将使用 mysql_store_result()函数。关于这两个函数的用法和区别，请参阅附录 G 里的相关讨论。

如果有连接参数没有在 connect()方法的参数里或连接属性所指定的选项文件里明确指定，DBI 将根据几个环境变量来为缺失的连接参数确定一个设置值：

- 如果数据源没有定义或者是一个空字符串，connect()将使用环境变量 DBI_DSN 的值作为数据源。
- 如果没有在数据源字符串里给出数据库驱动模块的名字，connect()将使用环境变量 DBI_DRIVER 所指定的数据库驱动模块。
- 如果 connect()方法的 user_name 和 password 参数没有定义（注意，不包括它们是空字符串的情况），就将使用环境变量 DBI_USER 和 DBI_PASS 的值作为用户名和口令。不过使用环境变量 DBI_PASS 的值作为口令是不安全的，在多用户系统上应该尽量避免这样做，因为其他用户往往能够通过一些系统监控命令查看到环境变量的明文内容。

如果 DBI 在检查过上述信息来源之后仍有一些连接参数不能确定，它就会使用一些默认值。比如说，如果是主机名无法确定，DBI 就将连接本地主机 localhost；如果是用户名无法确定，DBI 就将以你的登录名（如果你使用的是 Unix 系统）或“ODBC”（如果你使用的是 Windows 系统）为默认的用户名。但如果是口令无法确定，DBI 就没有默认值了，它将不发送口令。

■
```
$dbh = DBI->connect_cached ($data_source,
                            $user_name,
                            $password
                            [, \%attr]);
```

这个方法与 connect()只有一个区别：DBI 将把数据库句柄缓存起来。如果脚本在连接依然有效时又使用完全一样的连接参数发出了一个 connect_cached()调用，DBI 将返回那个被缓存起来的句柄，而不会再另外去创建一个新的连接。如果那个被缓存起来的数据库句柄已经失效，DBI 将建立一个新的连接并缓存和返回这个新句柄。

■ `@ary = DBI->data_sources ( $driver -name[, \%attr ] );`

返回一份现在能够通过给定驱动模块访问的数据库名单。与 MySQL 相对应的数据库驱动模块$driver_name 就是 mysql（注意要小写）。如果$driver_name 是 undef 或者是一个空字符串，DBI 将检查环境变量 DBI_DRIVER 以确定数据库驱动模块的名字。你可以使用可选的%attr 来提供连接参数。

很多 DBI 驱动模块上的 data_sources()调用都只能返回一个空名单或者一个不完整的名单。

■ `$drh = DBI->install_driver ( $driver_name );`

激活一个 DBD 级的驱动模块并返回一个驱动模块句柄，若无法找到指定的驱动模块，脚本将在显示一条出错信息后退出执行。与 MySQL 相对应的数据库驱动模块$driver_name 就是 mysql(注意要小写)。一般没有必要去亲自调用这个方法，因为 DBI 会在你发出 connect()调用的时候自动激活正确的数据库驱动模块。不过，如果你想通过 func()方法去执行一些管理性操作（参见 H.2.6 节）的话，install_driver()还是有用的。

■ `%drivers = DBI-> installed_drivers ();`

为已被加载到当前进程里的驱动程序返回一个以“驱动程序名/驱动程序句柄”对构成的散列。这个方法是从 DBI 1.49 版开始引入的。

H.2.2　数据库句柄方法

本节里的方法都需要使用一个数据库句柄来调用，在使用这些方法之前，应该先通过上一节介绍的 connect()或 connect_cached()或 clone()调用去获得一个这样的句柄。

在本节里，各方法的%attr 参数可以用来设定该方法的处理属性。(如果某个属性参数取值为 undef，则表示“不具备该属性”。）对与 MySQL 有关的 DBI 方法来说，最重要的属性是 PrintError 和 RaiseError。比如说，如果 RaiseError 属性当前处于禁用状态，下面这行代码将在处理特定语句期间激活它：如果在此期间发生了一个 DBI 错误，脚本就将自动退出执行：

```
$rows = $dbh->do ($statement, {RaiseError => 1});
```

H.4 节将对 PrintError、RaiseError 做详细的讨论。

❑ `$rc = $dbh->begin_work ();`

这个调用将禁用数据库句柄属性 AutoCommit，由此关闭自动提交模式。这将使我们能够执行一个事务。在下一个 commit()或 rollback()调用发生之前，AutoCommit 将一直保持在禁用状态；在下一个 commit()或 rollback()调用发生之后，AutoCommit 将自动恢复为启用状态。使用 begin_work()方法与以手动方式把 AutoCommit 属性设置为禁用状态是不同的；在后一种场合，AutoCommit 不会在 commit()或 rollback()调用发出之后自动恢复启用，必须手动重新启用。

如果成功禁用 AutoCommit，则 begin_work()返回 true；如果已经禁用，则返回 false。

- `$dbh2 = $dbh-> clone ([\%attr]);`

  复制现有连接$dbh 并返回一个新的数据库句柄。建立新连接时使用的参数和旧连接的一样。新给出的属性将加入到原有的属性当中，同名属性的旧值将被替换为新值。

- `$rc = $dbh->commit ();`

  提交当前事务，前提是 AutoCommit 处于禁用状态。若 AutoCommit 已启用，发出 commit()调用将没有任何效果并会导致一条警告信息。

- `$rc = $dbh->disconnect ();`

  断开与数据库句柄相关联的连接。如果某个连接在脚本退出执行时仍保持在连通状态，DBI 将自动断开这个连接并给出一条警告信息。

  DBI 没有对在事务处理的过程中发出的 disconnect()调用的行为作出定义。正确的做法是：如果交易尚未完成，就应该在发出 disconnect()调用之前先发出一个 commit()或 rollback()调用。

- ```
  $rows = $dbh->do ($statement
                  [, \%attr
                  [, @bind_values]]);
  ```

  对于 MySQL，如果你断开连接时没有提交正进行着的事务，服务器将回滚这个事务。

  准备并执行$statement 参数给出的语句。若调用成功，这个方法将返回修改的行数；若无法确定受影响的行数，则返回-1；若执行出错，将返回 undef。如果受影响的行数是零，这个方法的返回值将是字符串'0E0'。'0E0'在数值上下文里将被求值为数值 0，在布尔上下文里将被视为真。

  do()方法的主要用途是执行那些不会返回数据行的 SQL 语句 DELETE、INSERT、REPLACE 或 UPDATE 等。用 do()方法去执行一条 SELECT 语句是不恰当的。因为无法得到一个语句句柄，所以你将无法取回任何一个数据行。

  一般来说，在调用 do()方法的时候通常用不着传递什么属性，所以它的%attr 参数几乎总是被设置为 undef。@bina_value 参数对应着一个数组，它代表着一组将被绑定到占位符上的值。占位操符是一些我们在构造语句字符串时安排在里面的“?”字符。

  如果查询命令里没有占位符，%attr 参数和@bina_value 值都可以省略，如下所示：

  ```
  $rows = $dbh->do (
              "UPDATE member SET expiration = NOW() WHERE member_id = 39"
          );
  ```

  但如果语句里真的包含有占位符，%attr 参数就不允许省略，而且@bind_value 数组里的元素个数也必须与占位符个数相等。在下面这个调用里，因为语句字符串里有两个占位符，所以我们要把它的属性参数%attr 设置为 undef，然后再给出两个用来替换占位符的绑定值：

  ```
  $rows = $dbh->do ("UPDATE member SET expiration = ? WHERE member_id = ?",
                    undef,
                    "2007-11-30", 39);
  ```

- `$rv = $dbh-> get_info ($info_type);`

  返回一项关于 DBI 或驱动程序的描述性信息。

  ```
  my $version = $dbh->get_info (18); # get database version
  ```

  用来表明信息类型的$info_type 参数的可取值及其含义可以在 DBI 文档里查到：

  ```
  % perldoc DBI
  ```
  ```

- `$rc = $dbh->`**`ping`**` ();`

  在连接上服务器之后，如果脚本长时间没有进行数据库操作，服务器就会在一段倒计时时间结束后自动断开这个连接。ping()调用将重新建立这条连接。如果连接仍处于连通状态或者成功地重新连接上了服务器，ping()调用将返回真；否则，将返回假。

- `$sth = $dbh->`**`prepare`**` ( $statement [, \%attr ] );`

  对$statement 参数给出的查询命令进行预处理以便在今后执行。如果调用成功，prepare()将返回一个语句句柄；如果执行出错，则返回 undef。如果 prepare()调用成功，脚本就可以用它返回的语句句柄去发出 execute()调用以执行那条语句。

-
```
$sth = $dbh->prepare_cached ($statement
                             [, \%attr
                             [, $if_active]]);
```

  这个方法与 prepare()只有一个区别：DBI 将把语句句柄缓存起来。如果脚本在前一个 prepare_ cached()所返回的语句句柄依然有效的期间又使用同样的$statement 和%attr 参数发出了一个 prepare_cached()调用，DBI 将返回那个被缓存起来的语句句柄，而不会再另外去创建一个新的句柄。如果缓存的句柄依然有效，$if_active 参数将决定方法的行为。如果这个参数不存在或者值为 0，DBI 将调用 finish()并给出一条警告信息。如果$if_active 为 2，则 DBI 不会检查句柄是否有效；如果为 3，则缓存的句柄从缓存中移除，预处理一个新句柄并缓存。现有句柄不会改变，但不会缓存。

- `$str = $dbh->`**`quote`**` ($value [, $data_type]);`

  给字符串加上引号并对其中的 SQL 特殊字符进行转义。结果字符串可能用作语句中的一个数据值，你执行此语句时不会出现语句错误。比如说，字符串“I'm happy”将变成“I\'m happy”。如果$value 参数的取值是 undef，quote()将返回单词“NULL”。注意，返回的字符串都已经加上了引号，所以你在往语句字符串里插入这些值时就不应该再加引号了。

  不要使用 quote()方法对将要用来替换占位符的数据值进行预处理，DBI 会自动加上引号。

  MySQL 能够把语句里的字符串数据值转换为必要的数据类型，所以$data_type 参数通常用不着给出。不过，对于特定类型的值，需要给出一个$data_type 参数。比如说，可以用 DBI::SQL_INTEGER 来表明$value 参数代表的是一个整数。

- `$str = $dbh->`**`quote_identifier`**` ($name [, $name, ... [, \%attr ]]);`

  把给定的名字当做标识符并把它转换为一个引号型标识符。比如说，abc 将被转换为`abc`，a`c 将被转换为`a``c`。如果你给出了多个输入参数，这个方法将先给它们分别加上反引号，再用句号把它们串连在一起。这使我们可以构造出引号型的限定标识符来。比如说，quote_identifier('db', 'tbl', 'col')的执行结果将是`db`.`tbl`.`col`。

  这个方法和 quote()方法的用途是一样的，只不过针对的是诸如数据库名、数据表名、数据列名、索引名和各种假名之类的标识符，而不是针对数据值。在构造一条语句的时候，如果你引用的标识符包含空格或是其他通常不允许用于名字的特殊字符，这个方法就非常有用了。比如说，如果某个数据表的名字是“my table”（不包括中文引号），就不能像下面这样直接用在语句里，因为这个名字本身包含一个空格：

```
SELECT * FROM my table
```

  在 MySQL 里，只要把这样的名字用反引号括起来就可以在语句里引用了：

```
SELECT * FROM `my table`
```

在 DBI 里构造这样一条语句的时候，最稳妥的办法就是使用 quote_identifier()：

```
$stmt = "SELECT * FROM " . $dbh->quote_identifier ("my table");
```

- $rc = $dbh->**rollback** ();

  复原当前事务，前提是 AutoCommit 处于禁用状态。若 AutoCommit 处于启用状态，发出 rollback()调用将没有任何效果并会导致一条警告信息。

- $ary_ref = $dbh->**selectall_arrayref** ($statement
                                          [, \%attr
                                          [, @bind_values]]);

  执行$statement 参数所给出的数据库查询命令。这个方法的调用效果相当于连续发出 prepare()、execute()和 fetchall_arrayref()3 个调用。如果$statement 参数已经是一个语句句柄（也就是说，你在发出 selectall_arrayref()调用之前已经用 prepare()方法对该语句进行过预处理），selectall_arrayref()将省略预处理步骤。%attr 和@bind_values 参数的含义和用法与它们在 do()方法中的情况相同。

  返回值是一个数组引用，这个数组里的元素本身又是一些数组引用，它们所指向的数组分别对应着结果集里的一个数据行。如果结果集里没有数据行，返回的引用将指向一个空数组。

  如果发生错误，selectall_arrayref()将返回 undef，除非已取回部分结果集。如果发生了错误，selectall_arrayref()将把检索到的数据行返回给脚本。如果你想知道非 undef 返回值代表的是成功还是失败，可以去发出一个$dbh->err()调用或者去查看$DBI::err 属性。

- $hash_ref = $dbh->**selectall_hashref** ($statement,
                                          $key_col
                                          [, \%attr
                                          [, @bind_values]]);

  执行$statement 参数所给出的数据库查询命令。这个方法的调用效果相当于连续发出 prepare()、execute()和 fetchall_hashref()3 个调用。如果$statement 参数已经是一个语句句柄（也就是说，你在发出 selectall_hashref()调用之前已经用 prepare()方法对该语句进行过预处理），selectall_hashref()将省略预处理步骤。%attr 和@bind_values 参数的含义和用法与它们在 do()方法中的情况相同。

  返回值是一个散列值引用，这个散列值里的各个元素的键字是$key_col 参数指定的数据列的取值，它或者是数据库查询命令所选取的某个数据列的名称，或者是一个数据列编号（从 1 开始）。为避免散列值里的键同名而导致的数据行丢失，键数据列里的取值应该各不相同。如果结果集里没有数据行，散列将为空。然而，每个散列元素的值是一个散列引用，它包含结果集里的一行，并以语句所选取的列的名字为键。

  如果发生了错误，selectall_hashref()将返回 undef，除非已取回部分结果集。如果发生了错误，将把检索到的数据行返回给脚本。如果你想知道非 undef 返回值代表的是成功还是失败，可以去发出一个$dbh->err()调用或者去查看$DBI::err 属性。

- $ary_ref = $dbh->**selectcol_arrayref** ($statement,
                                          [\%attr
                                          [, @bind_values]]);

  执行$statement 参数所给出的数据库查询命令。这个方法的调用效果相当于连续发出 prepare()、execute()和一个数据行取回调用。如果$statement 参数已经是一个语句句柄

(也就是说，你在发出 selectcol_arrayref()调用之前已经用 prepare()方法对该语句进行过预处理)，selectcol_arrayref()将省略预处理步骤。%attr 和@bind_values 参数的含义和用法与它们在 do()方法中的情况相同。

如果调用成功，将返回一个数组引用，这个数组里的各个元素分别对应着结果集中数据行的第一个数据列。

如果发生了错误，selectcol_arrayref()将返回 undef，除非已取回部分结果集。如果发生了错误，将把检索到的数据行返回给脚本。如果你想知道非 undef 返回值代表的是成功还是失败，可以去发出一个$dbh->err()调用或者去查看$DBI::err 属性。

- @row_ary = $dbh->**selectrow_array** ($statement
  [, \%attr
  [, @bind_values]]);

执行$statement 参数所给出的数据库查询命令。这个方法的调用效果相当于连续发出 prepare()、execute()和 fetchrow_array()3 个调用。如果$statement 参数已经是一个语句句柄（也就是说，你在发出 selectrow_array()调用之前已经用 prepare()方法对该语句进行过预处理)，selectrow_array()将省略预处理步骤。%attr 和@bind_values 参数的含义和用法与它们在 do()方法中的情况相同。

如果在一个列表上下文里调用 selectrow_array()方法，这个方法将把结果集里的第一个数据行取回到一个数组并返回该数组；如果结果集里没有数据行可供返回或者是发生了错误，这个方法将返回一个空数组。如果在一个标量上下文里调用 selectrow_array()方法，将返回数组的一个元素；如果结果集里没有数据行可供返回或者是发生了错误，这个方法将返回 undef。返回哪个数组元素是无法预料的；请参阅关于 fetchrow_array()方法的条目里对这种行为的说明。

在列表上下文里，如果这个方法返回了一个空数组，你可以通过调用$dbh->err()方法或是查看$DBI::err 变量来区分那是因为结果集里没有数据行可供返回还是因为发生了错误。零值表明没有数据行可供返回。在标量上下文里，在没有发生错误的前提下，undef 返回值既有可能来自一个取值为 NULL 的数据列，也有可能是因为没有数据行可供返回。

- $ary_ref = $dbh->**selectrow_arrayref** ($statement
  [, \%attr
  [, @bind_values]]);

执行$statement 参数所给出的数据库查询命令。这个方法相当于连续发出 prepare()、execute()和 fetchrow_arrayref()3 个调用。如果$statement 参数已经是一个语句句柄(也就是说，你在发出 selectrow_arrayref()调用之前已经用 prepare()方法对该语句进行过预处理)，selectrow_arrayref()将省略预处理步骤。%attr 和@bind_values 参数的含义和用法与它们在 do()方法中的情况相同。

如果调用成功，将返回一个数组引用，这个数组对应着结果集里的第一个数据行；如果执行出错，将返回 undef。

- $hash_ref = $dbh->**selectrow_hashref** ($statement
  [, \%attr
  [, @bind_values]]);

执行$statement 参数所给出的数据库查询命令。这个方法相当于连续发出 prepare()、execute()和 fetchrow_hashref()3 个调用。如果$statement 参数已经是一个语句句柄（也

就是说，你在发出 selectrow_hashref()调用之前已经用 prepare()方法对该语句进行过预处理)，selectrow_hashref()将省略预处理步骤。%attr 和@bind_values 参数的含义和用法与它们在 do()方法中的情况相同。

如果调用成功，将返回一个散列值，这个散列值对应着结果集里的第一个数据行并以$statement 语句所选取的数据列的名称作为键。如果执行出错，则将返回 undef。

DBI 在自己的新版本里增加了一些用来获取数据库和数据表元数据的数据库句柄方法。这些方法包括 column_info()、foreign_key_info()、last_insert_id()、primary_key()、primary_key_info()、statistics_info()、table_info()、tables()、type_info()和 type_info_all()等。它们的详细情况可以在 DBI 文档中查到：

```
% perldoc DBI
```

不同的驱动程序对这些新方法的支持程度各不相同，其中一些方法本身还属于试验性质。就 MySQL 而言，你应该在你的 DBD::mysql 版本环境下试用这些方法，了解都有哪些已被实现、可以获得什么样的信息，等等。

H.2.3 语句句柄方法

本节里的方法都需要使用一个语句句柄来调用，在使用这些方法之前，应该先通过 prepare()或 prepare_cached()调用去获得一个这样的句柄。

- `$rc = $sth->`**`bind_col`**` ($col_num, \$var);`

  把 SELECT 查询所选取的某一个数据列绑定到一个 Perl 变量（注意，这个变量必须以一个引用的方式给出）上。$col_num 的取值范围是从 1 到这个查询所选取的数据列的个数。脚本每取回一个数据行，变量就会自动更新为指定数据列的新值。

  bind_col()调用应该发生在 execute 调用之后、开始取回数据行之前。

  如果$col_num 参数指定的数据列编号超出了这个范围，bind_col()调用将返回一个假值。

- `$rc = $sth->`**`bind_columns`**` (\$var1, \$var2, ...);`

  把预处理后的 SELECT 查询所选取的各个数据列依次绑定到由参数$var1、$var2 等变量。参见对 bind_col()方法的描述。与 bind_col()方法的情况类似，bind_columns()调用也应该发生在 execute()调用之后、开始取回数据行之前。

  如果参数的个数与查询命令所选取的数据列个数不匹配，bind_columns()调用将返回一个假值。

- `$rv = $sth->`**`bind_param`**` ($n, $value [, \%attr]);`

  `$rv = $sth->`**`bind_param`**` ($n, $value [, $bind_type]);`

  把一个数据值绑定到查询命令字符串里的一个占位符上，这个值将随着查询命令一起被发送到服务器去。占位符在查询命令字符串里表示为问号“?”。bind_param()调用应该发生在 prepare()调用之后、execute()调用之前。

  数据值$value 将被绑定到编号为$n 的那个占位符上，$n 的取值范围是从 1 到占位符的个数。如果需要绑定的数据值是 NULL，$value 就应该是 undef。

  %attr 或$bind_type 应该表明待绑定值的类型。在默认的情况下，DBI 会把被绑定数据视为一个 VARCHAR 值，所以非 NULL 值被绑定到语句时将加上引号。因为 MySQL 能够根据上下文把查询命令里的字符串数据值转换为必要的数据类型，所以这个方法里的%attr 和$bind_type 参数通常用不着给出。不过，这在有些上下文中会出问题。例如，LIMIT 子句的

任何参数必须是整数。下面两条$bind_param()调用都可以用来把被绑定数据限定为一个整数：

```
$rv = $sth->bind_param ($n, $value, { TYPE => DBI::SQL_INTEGER });
$rv = $sth->bind_param ($n, $value, DBI::SQL_INTEGER);
```

- $rv = $sth->**bind_param_array** ($n, $values [, \%attr]);

  $rv = $sth->**bind_param_array** ($n, $values [, $bind_type]);

  这个函数与 bind_param()类似，但它专门和预处理语句一起使用，以便用于 execute_array()的执行。$values 参数可以是一个数组引用，或一个标量值。对于数组引用，数组中的后续值用于语句的后续执行。对于标量值，每个语句都要重用这个值。
- $rows = $sth->**dump_results** ([$maxlen

  　　　　　　　　　　　　　　　[, $line_sep

  　　　　　　　　　　　　　　　[, $field_sep

  　　　　　　　　　　　　　　　[, $fh]]]]);

  取回语句句柄$sth 上的所有数据行，调用 DBI 工具函数 DBI::neat_list()对它们进行格式化，并把它们打印到参数$fh 指定的文件里去。这个方法的返回值是受其影响的数据行的个数。$max_len、$line_sep、$field_sep、$fh 等参数的默认值分别是：35、"\n"、"和 STDOUT。
- $rv = $sth->**execute** ([@bind_values]);

  执行预处理语句。对于 SELECT 类语句，如果调用成功，execute()将返回一个真值；如果执行出错，则返回一个假值。对于非 SELECT 类语句，如果调用成功，execute()将返回受影响的行数；若无法确定行数，则返回-1；若执行出错，将返回 undef。如果受其影响的数据行个数是零，这个方法的返回值将是字符串'0E0'。'0E0'在数值上下文里将被求值为数值 0，在布尔上下文里将被视为真。

  @bind_values 参数的含义和用法与它们在 do()方法中的情况相同。
- $rv = $sth->**execute_array** (\%attr [, @bind_values]);

  多次执行同一条预处理语句，执行次数将根据经@bind_values 参数传递的值的个数、此前通过调用 bind_param_array()方法绑定到该语句的值的个数，或者属性参数所指向的值来确定。
- $ary_ref = $sth->**fetch** ();

  fetch()是 fetchrow_arrayref()方法的一个别名。
- $ary_ref = $sth->**fetchall_arrayref** ([$slice_ref [, $max_rows]]);

  取回语句句柄$sth 上的所有数据行并返回一个数组引用，这个数组里的元素也是一些引用，它们依次指向结果集里的每一个数据行。如果结果集里没有数据行，fetchall_arrayref()返回的引用将指向一个空数组。否则，数组$ary_ref 里的各个元素将依次指向结果集里的每一个数据行。这些数据行引用的含义要根据你传递给这个方法的输入参数来确定。如果没有参数或者参数是一些数组下标值，这些数据行引用将指向一个由数据列值构成的数组。非空数组下标应该包含数组索引号，以选取特定的数据列。数组下标值将从 0 开始计算，因为它们给出的是某个 Perl 数组的下标，负下标值表示从数据行的尾端倒数计算。比如说，如果想把数据行的第一个和最后一个数据列提取出来，就需要使用下面这样的代码：
- $ary_ref = $sth->fetchall_arrayref ([0, -1]);

  如果参数是一些散列下标值，这些数据行引用指针将分别指向一个由数据列值构成的散列值，这些散列值的键字就是你要检索的数据列的名称。散列下标值的使用方法是：以数据列的名字作为散列键，以 1 作为键值，如下所示：

- $ary_ref = $sth->fetchall_arrayref ({id => 1, name => 1});

  如果想把所有的数据列全都取到一个散列值里去，就需要使用一个空散列值的引用作为参数，如下所示：

- $ary_ref = $sth->fetchall_arrayref ({});

  给定$max_rows 参数将限定提取的行数。此时，可以继续调用 fetchall_arrayref()，直到不再返回数据行。

  即使执行出错，fetchall_arrayref()也能把已经检索到的数据行返回给脚本。如果想知道是否发生过一个错误，可以发出一个$sth->err()调用或者去查看$DBI::err 属性。

- $hash_ref = $sth->**fetchall_hashref** ($key_col);

  取回结果集。如果调用成功，这个方法将返回一个散列值引用，这个散列值里的各个元素是结果集的每行，键字是$key_col 参数指定的数据列的取值，它或者是数据库查询命令所选取的某个数据列的名称，或者是一个数据列编号（从 1 开始）。为避免散列值里的键字同名而导致数据行丢失，键数据列里的取值应该各不相同。如果结果集里没有数据行，返回的引用将指向一个空散列表。然而，这个散列值里的各个元素的值则又是一个散列值引用，分别对应着结果集里的一个数据行，并以语句选取的数据列的名字作为各散列元素的键字。

  如果是因$key_col 参数不合法而导致执行出错，fetchall_hashref()将返回 undef；否则，即使执行出错，它也能把检索到的数据行全部返回给脚本。如果想知道非 undef 返回值代表的是成功还是失败，可以去发出一个$sth->err()调用或者去查看$DBI::err 属性。

- @ary = $sth->**fetchrow_array** ();

  在列表上下文里，如果调用成功，这个方法将返回一个数组，包含结果集里的下一个数据行；如果已经到达结果集里的最后一个数据行或执行出错，则返回空数组。如果想知道是"正常到达结果集的末尾"还是"发生了一个错误"，可以发出一个$sth->err()调用或者去查看$DBI::err 属性。零值代表"正常到达结果集的末尾"。

  在标量上下文里，如果调用成功，这个方法将返回数组的某一个元素；如果已经到达结果集里的最后一个数据行或执行出错，则将返回 undef。但对于返回哪个数据列，并没有明确的规定，只有在结果集里只包含一个数据行的时候才是准确的。同样，如果执行没有出错，undef 返回值的含义将难以确定：既有可能是数据列的取值就是 NULL，也有可能是"没有返回数据行"。

- $ary_ref = $sth->**fetchrow_arrayref** ();

  如果调用成功，则返回一个数组引用，包含结果集里下一个数据行的列值；如果已经到达结果集里的最后一个数据行或执行出错，则返回 undef。

  如果想知道这个方法的 undef 返回值的含义是"正常到达结果集的末尾"还是"发生了一个错误"，可以发出一个$sth->err()调用或者去查看$DBI::err 属性。零值代表"正常到达结果集的末尾"。

- $hash_ref = $sth->**fetchrow_hashref** ([$name]);

  如果调用成功，这个方法将返回一个散列引用，包含结果集里下一个数据行的数据列值；如果已经到达结果集里的最后一个数据行或执行出错，则将返回 undef。散列元素的键字是数据列的名字，键值则是数据列里的数据值。

  $name 参数用来控制散列键的字母大小写情况。它的默认值是"NAME"（意思是以数据库查询命令中给出的数据列名称作为键）。把$name 参数设置为"NAME_lc"或"NAME_uc"将把构成散列键的字符全都强制转换为小写或大写字母。（FetchHashKeyName 属性也是控制散列键字大

小写情况的办法之一，详细情况请参见 H.4 节。）

如果想知道这个方法的 undef 返回值的含义是“正常到达结果集的末尾”还是“发生了一个错误”，可以发出一个$sth->err()调用或者去查看$DBI::err 属性，零值代表“正常到达结果集的末尾”。

- $rc = $sth->**finish** ();

  释放与语句句柄相关联的全部资源。我们通常用不着亲自调用这个方法，因为那些用来取回数据行的方法会在到达结果集里的最后一个数据行时隐含地调用它。但如果你只取回了结果集的一部分，就应该明确地调用 finish()，通知 DBI 说你已经完成了这语句句柄上的数据行取回工作。发出 finish()调用之后，有关的语句属性都将失效。又因为那些用来取回数据行的方法会在检测到已经到达结果集末尾的时候隐含地调用 finish()，所以你最好别耽误工夫，在调用完 execute()方法之后立刻访问有关属性。

- $rv = $sth->**rows** ();

  返回一个数据行计数值，即有多少个数据行受到了与$sth 相关联的那条语句的影响；如果执行出错，则将返回−1。rows()方法多用来检查 UPDATE 或 DELETE 等不会返回数据行的语句的执行情况。对 SLELCT 类语句最好不要使用 rows()方法；应该一边从结果集里提取数据行，一边统计数据行的个数。

H.2.4　通用句柄方法

本节里的方法不需要通过特定类型的句柄来调用。你可以使用驱动模块句柄、数据库句柄或语句句柄来调用它们。

- $rv = $h->**err** ();

  返回最近一次调用的 DBI 操作的数值出错代码。就 MySQL 而言，它就是 MySQL 服务器返回的错误代号。如果这个返回值是 0 或 undef，就表明调用执行过程没有出错；如果为空，表明“成功”，此时 errstr()返回额外信息。

- $str = $h->**errstr** ();

  返回最近一次调用的 DBI 操作的字符串出错信息。就 MySQL 而言，它就是 MySQL 服务器返回的出错信息。如果这个返回值是一个空字符串或 undef，就表明调用执行过程没有出错。

- DBI->**trace** ($trace_level [, $trace_filename]);

  $h->**trace** ($trace_level [, $trace_filename]);

  设置一个跟踪级别。跟踪机制能够让我们掌握 DBI 操作的执行情况。跟踪级别从 0（关闭）到 9（信息量最大）共分 10 档。你可以跟踪脚本里的所有 DBI 操作（办法是把 trace()作为一个 DBI 类方法来调用），也可以只跟踪某个特定的句柄，如下所示：

```
DBI->trace (2);        # Turn on global script tracing
$sth->trace (2);       # Turn on per-handle tracing
```

  如果想跟踪自己运行的所有 DBI 脚本，那么设置 DBI_TRACE 环境变量。

  在默认的情况下，调试信息将被发送到 STDERR。但如果指定了$filename 参数，就可以把调试信息重定向到这个文件里去。输出信息将被追加在这个文件末尾而不会覆盖掉它原有的内容。特殊文件名 STDOUT 和 STDERR 分别表示标准输出和标准错误输出。

  注意，来自所有被跟踪句柄的调试信息将都被写入同一个文件。如果你给出了一个文件名，那么所有的调试信息将都被写入这个文件；如果你没有给出这个参数，那么所有的调试信息将都

被送往 STDERR 设备。

- DBI->**trace_msg** ($str [, $min_level]);
  $h->**trace_msg** ($str [, $min_level]);
  如果把 trace_msg()当做一个类方法来调用（DBI->trace_msg()），那么，只要跟踪机制已经在 DBI 级别上被激活，它就会把调试信息$str 写到调试输出设备去。如果把 trace_msg()当做一个句柄方法来调用（$h->trace_msg()），那么，只有在你对这个句柄做过 trace()调用或者在 DBI 级别启用了跟踪机制的情况下，它才会把调试信息$str 写到调试输出设备去。如果还给出了一个$min_level 参数，那么只有在跟踪级别至少是$min_level 参数的值时，调试信息$str 才会被写到调试输出设备上。

H.2.5 用来执行MySQL数据库管理操作的方法

本节将介绍如何使用 DBI 提供的 func()方法来直接完成一些数据库驱动模块级的操作。注意，func()方法与数据库中的存储例程（stored procedure）毫不相干。DBI 目前还不支持存储例程方法。

```
$rc = $drh->func ("createdb", $db_name,
                  $host_name, $user_name, $password, "admin");
$rc = $drh->func ("dropdb", $db_name,
                  $host_name, $user_name, $password, "admin");
$rc = $drh->func ("reload",
                  $host_name, $user_name, $password, "admin");
$rc = $drh->func ("shutdown",
                  $host_name, $user_name, $password, "admin");

$rc = $dbh->func ("createdb", $db_name, "admin");
$rc = $dbh->func ("dropdb", $db_name, "admin");
$rc = $dbh->func ("reload", "admin");
$rc = $dbh->func ("shutdown", "admin");
```

func()方法既可以通过一个数据库驱动模块句柄来调用，也可以通过一个数据库句柄来调用。因为驱动模块句柄并不与某个已经打开的连接相关联，所以如果想以这种方式来调用 func()方法，就必须给出相应的主机名、用户名、口令等参数才行，func()方法需要使用这些参数去连接数据库服务器。如果你是通过一个数据库句柄来调用 func()方法的，就不需要给出这些参数了。如有必要，可以通过下面这个办法来获得一个驱动模块句柄：

```
$drh = DBI->install_driver ("mysql");       # "mysql" must be lowercase
```

func()方法能够识别的操作动作有以下几种。

- createdb
  以$db_name 参数给出的名字创建一个数据库。你必须有这个数据库的 CREATE 权限才能执行这个操作。
- dropdb
  丢弃（即删除）$db_name 参数指定的数据库。你必须有这个数据库上的 DROP 权限才能执行这个操作。
  请三思而后行；在执行了数据库丢弃操作之后，数据库里的数据就找不回来了。
- reload
  重新加载权限表。只有在你使用 DELETE、INSERT 或 UPDATE 等语句而非 GRANT、REVOKE 等语句去修改权限表的内容时才需要发出这个调用。你必须有这个数据库上的 RELOAD 权

限才能执行这个操作。

- shutdown

  关闭服务器。必须有这个数据库上的 SHUTDOWN 权限才能执行这个操作。

注意，在上述操作中，只有 shutdown 是唯一能够通过正常的 DBI 查询处理机制来进行的 func() 动作。其他动作最好用 CREATE DATABASE、DROP DATABASE 或 FLUSH PRIVILEGES 等 SQL 语句而不是 func() 调去完成。

H.3 DBI 工具函数

DBI 还准备了一些辅助工具例程，用于完成数据值的测试和打印输出工作。这些函数必须以 DBI::func_name() 而不是 DBI->func_name() 的形式去调用。

- @bool = DBI::**looks_like_number** (@ary);

  对数组中的各个元素看起来是否像一个数字进行测试。这个方法的返回值是一个数组，该数组里的每一个成员对应着一个元素。如果数组中的某个元素看起来像一个数字，数组中与之对应的元素就是一个真值；如果不像，返回一个假值；如果某个元素未定义或者为空，将返回一个 undef。

- $str = DBI::**neat** ($value [, $maxlen]);

  对$value 参数给出的数据值进行格式化并返回结果字符串。如果$value 是一个字符串，则给它加上引号；如果它是一个数值，则不加引号。（注意：用引号里括起来的数值将被视为一个字符串。）如果它是一个未定义值，则返回 undef；如果是非打印字符，则返回“.”。我们来看一个例子：

```
for my $val ("a", "3", 3, undef, "\x01\x02")
{
  print DBI::neat ($val), "\n";
}
```

  如果执行上面这个循环，你将得到如下所示的输出：

```
'a'
'3'
3
undef
'..'
```

  $maxlen 参数控制着结果的最大长度；如果结果的长度超过了$maxlen，DBI 就将把它截短为 $maxlen-4 个字符，并在其后加上“...”。如果参数$maxlen 等于 0、undef 或者没有被给出，它的默认值将等于$DBI::neat_maxlen 的当前值，$DBI::neat_maxlen 本身的默认值是 400。

  不要用 neat() 函数来构造查询命令字符串，若要对将被插入查询命令字符串的数据值添加引号或者进行转义处理，应该使用使用占位符或者通过 quote() 调用去完成。

- $str = DBI::**neat_list** (\@ary [, $maxlen [, $sep]]);

  调用 neat() 函数对数组@ary 中的各个元素依次进行处理，再用分隔符$sep 把它们合并起来并返回为一个字符串。

  $maxlen 参数将被传递到 neat() 函数，所以它只对各参数有影响，最终的结果字符串不受这个参数的影响。

  如果$sep 参数没有给出，它的默认值将是", "（一个逗号加一个空格）。

H.4 DBI 属性

DBI 还提供有多种层次的属性信息。大多数属性被关联到数据库句柄或者语句句柄，但不允许同时关联到这两种句柄上，而有些属性（比如 PrinrError 或 RaiseError）却可以。一般说来，每个句柄都有它自己的属性，但那些用来存放出错信息的属性（比如 err 和 errstr）却是“动态”的，它们被关联到你最近一次使用的那个句柄。

被传递给 connect()或 connect_cached()方法的属性将成为这两个方法所返回的结果数据库句柄的一部分。

H.4.1 数据库句柄属性

本节里的 DBI 属性都是与数据库句柄相关联的。

- $dbh->{**AutoCommit**};

  把这个属性设置为真或假，就可以激活或者禁用自动提交模式，它默认设置是真。把 AutoCommit 属性设置为假，将开始执行事务，如果这些语句全都执行成功，我们就可以发出一个 commit()调用来提交这次事务；只要这些语句里有一条没有执行成功，就应该发出一个 rollback()调用来中断此事务。另请参见本附录前面对数据库句柄方法 begin_work()的描述。

- $dbh->{**Statement**};

  这个属性的值是最近一次通过给定数据库句柄传递给 prepare()方法的语句字符串。

H.4.2 通用句柄属性

下面这些属性既可以被关联到某个具体的句柄，也可以出现在属性参数%attr 里以影响该方法的操作行为。

- $h-> {**ChopBlanks**};

  把这个属性设置为“真”或“假”将决定各数据行取回方法是否会把字符数据列值的尾缀空格去除掉。ChopBlanks 属性在绝大多数数据库驱动程序中的默认设置都是“假”。

- $h-> {**FetchHashKeyName**};

  fetchrow_hashref()以及其他几个调用 fetchrow_hashref()的方法会把从结果集里取回的数据行返回为一些散列值，FetchHashKeyName 属性控制着这些散列值里键的字母大小写情况。这个属性的默认设置值是"NAME"（意思是以 SELECT 语句中给出的数据列名称作为键），它设置为"NAME_lc"或"NAME_uc"将把那些由数据列名称充当的散列键都强制转换为小写或大写字母。这个属性只适用于数据库句柄和数据库驱动模块句柄。

- $h-> {**HandleError**};

  这个属性是用来进行出错处理的。它可以被设置为一个出错时调用的例程的引用，DBI 将在调用 RaiseError 和 PrintError 等常规出错处理例程之前先调用它。如果这个例程的返回值是真，DBI 将跳过 RaiseError 或 PrintError 等常规出错处理例程；否则，它将照常执行。（当然，你编写的这个例程完全可以终止脚本的执行而不是返回。）

  DBI 将向出错处理例程传递 3 个参数——出错消息的文本、出错时脚本正在使用的那个 DBI 句柄、执行出错的那个方法所返回的第一个值。

- $h-> {**PrintError**};

  如果这个属性被设置为真，与 DBI 有关的执行出错就会导致一条警告消息被打印出来。

PrintError 默认设置值是假。这个属性对执行出错的 DBI 方法的返回值并没有影响，它只决定它们是否需要在返回之前打印一条出错消息。

- $h-> {**RaiseError**};
  如果这个属性被设置为真，与 DBI 有关的执行出错就会导致脚本自动结束运行。这通常会让脚本终止，除非它捕捉到异常。这个属性的默认设置值是假。
- $h-> {**ShowErrorStatement**};
  如果这个属性被设置为真，DBI 就会把执行出错的那条数据库查询命令的文本追加在出错消息的末尾一起打印出来。Show Error Statement 的默认设置值是假。这个属性的效力只限于各语句句柄和 prepare()与 do()方法。
- $h-> {**TraceLevel**};
  设置或获取指定句柄上的跟踪级别。这个属性相当于 trace()方法的一个替代品。

H.4.3 MySQL特有的数据库句柄属性

本节里的属性都是 DBI MySQL 驱动模块 DBI::mysql 所特有的。正如大家将会看到的那样，这些属性分别对应着 MySQL C API 里的某个函数。C 函数的详细情况请参见本书的附录 F。

- $rv = $dbh->{**mysql_auto_reconnect**};
  驱动程序是否会在连接意外断开时自动重建与服务器的连接。这种自动重建连接功能通常是默认禁用的，但在你设置了 GATEWAY_INTERFACE 或 MOD_PERL 环境变量的情况下将默认激活。如果禁用了 AutoCommit 属性，mysql_auto_reconnect 属性的设置情况将被忽略，驱动程序不会自动重建连接。
- $hash_ref = $dbh->{**mysql_dbd_stats**};
  返回包含驱动块统计值的散列引用。当前这个散列有两个键，auto_reconnects_ok 和 auto_reconnects_failed，表明驱动程序重新连接服务器的成功次数和失败次数。
- $rv = $dbh->{**mysql_errno**};
  返回最近一次发生的错误的出错代码。这个属性与 C API 中的 mysql_errno()函数功能相当。
- $str = $dbh->{**mysql_error**};
  返回最近一次发生的错误的出错字符串。这个属性与 C API 中的 mysql_error()函数功能相当。
- $str = $dbh->{**mysql_hostinfo**};
  返回一个描述给定连接的字符串。这个属性与 C API 中的 mysql_get_host_info()函数功能相当。
- $str = $dbh->{**mysql_info**};
  返回影响多个数据行的语句的信息。这个属性与 C API 中的 mysql_info()函数功能相当。
- $rv = $dbh->{**mysql_insertid**};
  返回与$dbh 相关联的数据库连接上最近一次生成的 AUTO_INCREMENT 值。这个属性与 C API 中的 mysql_insert_id()函数功能相当。
- $rv = $dbh->{**mysql_protoinfo**};
  返回给定连接上所使用的客户/服务器协议版本，这个属性与 C API 中的 mysql_get_proto_info()函数功能相当。
- $rv = $dbh->{**mysql_server_prepare**};
  如果服务器端语句预处理功能已被激活，这个属性的值将是“真”；如果语句预处理是在客户

端模拟出来的，这个属性的值将是“假”。

你可以通过对这个属性赋值为给定数据库句柄$dbh 所创建的语句句柄激活或禁用服务器端预处理语句执行功能：

```
$dbh->{mysql_server_prepare} = 1; # enable server-side preparation
$dbh->{mysql_server_prepare} = 0; # disable server-side preparation
```

关于使用 mysql_server_prepare 属性所需要的 DBD::mysql 版本的信息，见本附录开头的概述部分。

- $str = $dbh->{**mysql_serverinfo**};

  一个描述服务器版本的字符串，例如"5.0.60-debug-log"。这个字符串值由一个版本号和一个或多个可选的后缀构成。这个属性返回的信息与 C API 函数 mysql_get_server_info()和 SQL 函数 VERSION()返回的完全一样。对各种后缀值的说明见附录 C 对 VERSION()函数的描述。

- $str = $dbh->{**mysql_stat**};

  返回一个字符串，里面包含着服务器的运行状态信息。这个属性与 C API 中的 mysql_stat()函数功能相当。

- $rv = $dbh->{**mysql_thread_id**};

  返回与$dbh 相关联的数据库连接的连接 ID。这个属性与 C API 中的 mysql_thread_id()函数和 CONNECTION_ID()功能相当。

- $rv = $dbh->{**mysql_use_result**};

  在检索结果集时是使用 C API 里的 mysql_store_result()函数还是使用 mysql_use_result()函数。请参阅第 H.4.5 节对与此有关的语句句柄属性的描述。

H.4.4 语句句柄属性

语句句柄属性大都只适用于 SELECT 查询（或者其他与 SELECT 类似的查询），而且，在你调用 prepare()方法获得一个语句句柄并通过这个句柄发出 execute()调用之前，这些属性往往不可用。此外，因为 finish()调用会使大多数语句句柄失效，所以在发出 finish()调用之后（或者在已经到达结果集里的最后一个数据行之后——因为那些数据行取回函数在到达结果集末尾时会隐含地发出一个 finish()调用）再去访问语句句柄属性是不明智的。

很多语句句柄属性的值是一个数组引用，这个数组里的元素分别对应着结果集里的一个数据行，而这个数组里的元素个数将由$sth->{ NUM_OF_FIELDS }属性值给出。假设语句属性 stmt_attr 就是一个这样的数组引用，那么，如果你需要把这个数组当做一个整体来访问，可以使用@{$sth->{stmt_att}}形式的语法；如果需要对这个数组里的各个元素依次处理，可以使用一个如下所示的循环语句：

```
for (my $i = 0; $i < $sth->{NUM_OF_FIELDS}; $i++)
{
    my $value = $sth->{stmt_attr}->[$i];
}
```

此外，NAME_hash、NAME_lc_hash 和 NAME_uc_hash 等几个属性返回散列引用，可以用一个如下所示的循环语句来遍历其中的各个元素：

```
foreach my $key (keys (%{$sth->{stmt_attr}}))
{
    my $value = $sth->{stmt_attr}->{$key};
}
```

- $ary_ref = $sth->{**NAME**};
  这个属性返回的引用指向一个字符串数组,这个数组的各个元素依次给出了结果集里各个数据列的名称，它们的大小写情况与你在 SELECT 语句里所给出的一致。
- $ary_ref = $sth->{**NAME_hash**};
  这个属性返回的引用指向一个由字符串构成的散列值，元素就是结果集里各个数据列的名称，它们的大小写情况与你在 SELECT 语句里所给出的一致。散列元素的值是与之对应的数据列在结果集数据行里的先后顺序（从 0 开始编号)。
- $ary_ref = $sth->{**NAME_lc**};
  类似于 NAME 属性，但数据列的名称将全部返回为小写的字符串。
- $ary_ref = $sth->{**NAME_lc_hash**};
  类似于 NAME_hash 属性，但数据列的名称将全部返回为小写的字符串。
  NAME_lc_hash 属性最早出现于 DBI 1.20 版本。
- $ary_ref = $sth->{**NAME_uc**};
  类似于 NAME 属性，但数据列的名称将全部返回为大写的字符串。
- $ary_ref = $sth->{**NAME_uc_hash**};
  类似于 NAME_hash 属性，但数据列的名称将全部返回为大写的字符串。
- $ary_ref = $sth->{**NULLABLE**};
  这个属性将返回一个数组引用，各个元素表明与之对应的数据列是否允许使用 NULL 值，它们的取值可以是 0 或空字符串（不允许使用 NULL 值)、1（允许使用 NULL 值）或 2（不详)。
- $rv = $sth->{**NUM_OF_FIELDS**};
  结果集里的数据列个数。对于非 SELECT 语句，这属性的返回值将是 0。
- $rv = $sth->{**NUM_OF_PARAMS**};
  prepare()调用所处理的查询命令里有多少个占位符。
- $ary_ref = $sth->{**PRECISION**};
  这个属性将返回一个数组引用，该数组的各个元素表明了结果集里的各个数据列的精度。DBI 使用的是 ODBC 对“精度”这个词的解释，它指的是 MySQL 数据列的最大宽度。对于数值数据列，这个“精度”其实只是它的显示宽度。对于字符串数据列，这个“精度”就是数据列最大长度，以字节而非字符计算。
- $ary_ref = $sth->{**SCALE**};
  这个属性将返回一个数组引用，该数组的各个元素表明了结果集里的各个数据列的范围。DBI 使用的是 ODBC 对“范围”这个词的解释，它指的是 MySQL 浮点数据列中小数点后面的位数。对于其他类型的数据列，这个属性所返回的“范围”值将等于 0。
- $str = $sth->{**Statement**};
  与语句句柄$sth 相关联的数据库查询语句的文本，即 prepare()调用所看到的、尚未进行占位符替换的那个查询命令字符串。
- $ary_ref = $sth->{**TYPE**};
  这个属性将返回一个数组引用，该数组的各个元素表明结果集里的各个数据列的类型。这个属性的返回值里包含的是数据类型编号，而 mysql_type 属性的返回值里则仍包含着 MySQL 专用的数据类型编号。

H.4.5 MySQL特有的语句句柄属性

本节里的属性都是 DBI MySQL 驱动模块 DBI::mysql 所特有的。它们当中的绝大多数都应该被当做只读属性来使用，并应该在 execute()调用完成之后再去访问。但 mysql_use_result 属性却是个例外，它应该在 prepare()调用之后、execute()调用之前设置，如下所示。

- $rv = $sth->{**mysql_insertid**};
  与语句句柄$sth 相关联的连接上最近一次生成的 AUTO_INCREMENT 值。
- $ary_ref = $sth->{**mysql_is_auto_increment**};
  这个属性将返回一个数组引用，该数组的各个元素表明结果集里的各个数据列是否是一个 AUTO_INCREMENT 数据列。
- $ary_ref = $sth->{**mysql_is_blob**};
  这个属性将返回一个数组引用，该数组的各个元素表明结果集里的各个数据列是否是一个 BLOB。对于 TEXT 数据列，数组中的值为真。
- $ary_ref = $sth->{**mysql_is_key**};
  这个属性将返回一个数组引用，该数组的各个元素表明结果集里的各个数据列是否是某个键的组成部分。
- $ary_ref = $sth->{**mysql_is_num**};
  这个属性将返回一个数组引用，该数组的各个元素表明结果集里的各个数据列是否是一个数值。
- $ary_ref = $sth->{**mysql_is_pri_key**};
  这个属性将返回一个数组引用，该数组的各个元素表明结果集里的各个数据列是否是某个 PRIMARY KEY 的组成部分。
- $ary_ref = $sth->{**mysql_length**};
  类似于 PRECISION 属性。
- $ary_ref = $sth->{**mysql_max_length**};
  这个属性将返回一个数组引用，该数组的各个元素给出了结果集各数据列里的数据值的实际最大长度。
- $rv = $sth->{**mysql_server_prepare**};
  如果启用服务器端语句预处理，则为 true，如果语句预处理是在客户端模拟的，则为 false。使用 mysql_server_prepare 属性需要的 DBI::mysql 版本信息，请参见本附录的介绍。
- $ary_ref = $sth->{**mysql_table**};
  这个属性将返回一个数组引用，该数组的各个元素表明各个数据列来自哪一个数据表。如果结果集里的某个数据列是某表达式的计算结果，对应元素将是一个空字符串。
- $ary_ref = $sth->{**mysql_type**};
  这个属性将返回一个数组引用，该数组的各个元素给出了结果集里的各个数据列的 MySQL 专用数据类型编号。
- $ary_ref = $sth->{**mysql_type_name**};
  这个属性将返回一个数组引用，该数组的各个元素给出了结果集里的各个数据列的 MySQL 专用数据类型名称。
- $rv-> $sth-> {**mysql_use_result**};
  在检索结果集时是使用 C API 里的 mysql_store_result()函数还是使用 mysql_use_

result()函数。在默认的情况下，mysql_use_result 属性的值是 0，所以 DBD::mysql 将使用 mysql_store_result()函数来检索结果集。如果你把 mysql_use_result 属性设置为 1，DBD::mysql 将使用 mysql_use_result()函数。关于这两个函数的用法和区别，请参阅附录 G 里的相关讨论。

请注意，激活 mysql_use_result 属性将导致某些语句句柄属性如 mysql_max_length 失效。它还会导致 rows()方法失效，你只能另想办法在取回数据行时对它们的个数进行统计了。

如果你打算设置 mysql_use_result 属性，必须在调用 prepare()方法之后、但在调用 execute()方法之前进行：

```
$sth = $dbh->prepare ($stmt_str);
$sth->{mysql_use_result} = 1;
$sth->execute();
```

或者像下面这样做：

```
$sth = $dbh->prepare ($stmt_str, { mysql_use_result => 1 });
```

- $rv-> $sth-> {**mysql_warning_count**};

  在执行这条语句期间生成的警告消息的个数。

  这个属性是从 DBD::mysql 4.004 版开始引入的。

H.4.6　动态属性

下面这些属性只与你最近一次使用的句柄（在以下内容里，我将使用$h 来代表这个句柄）相关联。如果你想正确地获得这些属性所返回的信息，就应该在调用会设置这些属性的句柄方法之后、在调用另一个会重新设置这些属性的句柄方法之前去访问它们。

- $rv = **$DBI::err** ;

  与调用$h->err()方法的效果相同。

- $str = **$DBI::errstr** ;

  与调用$h->errstr()方法的效果相同。

- $rows = **$DBI::rows** ;

  与调用$h->rows()方法的效果相同。

H.5　DBI 环境变量

表 H-2 列出了几个常用的环境变量。除 DBI_TRACE 外，这些环境变量都是 connect()方法会用到的。DBI_DRIVER 还可以用在 data_source()方法里。而 DBI_TRACE 是用在 tarce()方法里的。

表 H-2　DBI 环境变量

变　量　名	含　　义
DBI_DRIVER	DBD级的驱动模块名称（对应于MySQL的驱动模块是"mysql"）
DBI_DSN	数据源名称
DBI_PASS	口令
DBI_TRACE	跟踪级别 和/或 跟踪输出文件
DBI_USER	用户名

附录 I PHP API 指南

本附录将介绍使用 PDO 扩展来编写能够与 MySQL 交互的 PHP 脚本时需要用到的 API。这个 API 由一组用来与 MySQL 服务器通信和访问数据库的类和方法构成。

本附录中的例子只是一些简略的代码片段。完整的客户端代码及其编写指令请参见第 9 章。PHP 手册可从 PHP 网站 http://www.php.net/获取。

本附录还涉及 PDO 常数，对于 PHP 5.1 及以上版本，使用的是类常量符号，如 `PDO::FETCH_NUM`。在 PHP 5.0 中，使用的则是全局常量符号，如 `PDO_FETCH_NUM`。

I.1 编写 PHP 脚本

PHP 脚本本身是一些普通的文本文件，但它的内容却是 PHP 代码和非 PHP 代码（如 HTML）的一个混合体。PHP 脚本以解释方式执行，它们生成的输出就是一个将被发送给客户的 Web 页面。非 PHP 代码将不加解释地原样发送出去，其中的 PHP 代码则会被解释执行并被替换为这段代码所产生的输出。

PHP 将以文本复制模式里开始解释一个脚本文件。利用 PHP 代码的起始标记和结束标记时，你可以在一个脚本文件里在 HTML 模式和 PHP 代码执行模式之间切换任意。PHP 能够识别的标记共有 4 种，但它们当中有些需要在事先明确启用才能使用。启用办法之一是在 PHP 的初始化文件 php.ini 里进行设置。这个文件的存放位置会随着系统的不同而有所差异。在许多 Unix 系统上，它通常在/usr/lib 或/usr/local/lib 子目录里。在 Windows 系统上，它通常在 PHP 安装目录里。

PHP 能够识别以下几种标记。

- 默认样式，使用“`<?php`”和“`?>`”标记，如下所示：

```
<?php print ("Hello, world."); ?>
```

- 简约样式，使用“`<?`”和“`?>`”标记，如下所示：

```
<? print ("Hello, world."); ?>
```

  如果你只是想把某个表达式的结果打印出来，还可以省略 `print` 命令并将这对标记写成更简单的“`<?=`”和“`?>`”形式，如下所示：

```
<?= "Hello, world." ?>
```

  简约样式可以通过 PHP 初始化文件里的下面这条指令启用：

```
short_open_tag = On;
```

- Active Server Page 兼容样式，使用“<%”和“%>”标记，如下所示：

```
<% print ("Hello, world."); %>
```

如果只是想把某个表达式的结果打印出来，还可以省略 print 命令并这将对标记写成更简单的“<%=”和“%>”形式，如下所示：

```
<%= "Hello, world." %>
```

ASP 兼容样式可以通过 PHP 初始化文件里的下面这条指令启用：

```
asp_tags = On;
```

- 如果使用的 HTML 编辑器不支持上述风格的标记，可以使用<script>和</script>标记，如下所示：

```
<script language="php"> print ("Hello, world."); </script>
```

I.2　PDO 类

本附录将讨论 PDO 扩展中的以下几个类。

- PDO。它是基础类。这个类的构造器用来连接数据库服务器。它将返回一个数据库句柄对象，而程序员需要通过这个句柄的方法与数据库服务器打交道。
- PDOStatement。这是一个语句句柄类，该类对象由一个 PDO 对象的 query()或 prepare()方法返回。语句句柄向程序员提供了访问一条给定语句的执行结果（语句元数据、结果集内容，等等）的手段。
- PDOException。这是 PDO 中的出错处理类。当脚本代码因为发生了一个 PDO 错误而抛出一个异常时，我们可以通过这个类提供的方法获得诊断信息。

I.3　PDO 方法

本节将介绍可用的 PDO 方法，这些方法按照它们所关联的类来划分。某些特定的对象名会反复出现在本附录对各种方法的介绍当中，它们有着如下所示的固定含义。

- 数据库句柄方法统一使用一个$dbh 对象来调用。该对象通过调用 PDO 类的构造器 new PDO{}获得。
- 语句句柄方法统一使用一个$sth 对象来调用。该对象由$dbh->prpare()或$dbh->query()方法返回。
- 异常对象统一用$e 来代表。

在描述各个方法时，我们将注明其返回值和输入参数的数据类型。如果某个值会根据具体用途的不同而有不同的数据类型，我们将使用 mixed 作为它的类型以表明这种情况。

有很多方法会返回一个用以表明本次调用成功或失败的值。在没有启用 PDO 异常捕获机制的情况下，这种返回值有着很重要的意义，应该及时测试以决定下一步该如何行动。如果启用了 PDO 异常捕获机制，方法执行过程中发生的错误将导致 PDO 抛出一个 PDOException 对象，该对象可以用一个 try/catch 语法构造来捕获。（请参阅 I.3.3 节。）

语法描述中的方括号（[]）表示该参数是可选的。如果某个可选参数的后面带有= *value* 字样，其含义是：如果某个方法在调用时省略了这个参数，*value* 将是它的默认值。

本附录中的绝大多数示例代码都把各种消息和查询结果输出为普通文本。我们这么做的目的只是

为了让那些示例代码更加简明易读。对于一个将在 Web 环境里执行的脚本，应该用 htmlspecialchars()方法对它的输出内容进行编码，以避免因为其中包含诸如“<”、“>”或“&”之类的 HTML 特殊字符而导致异常。

在接下来的内容里，“SELECT 语句”代表着会返回一些数据行的任何语句，这包括 SELECT 语句本身以及 DESCRIBE、EXPLAIN、SHOW 等语句。

I.3.1 PDO类方法

PDO 类的方法按其用途可以大致分为连接数据库服务器、对 SQL 语句进行预处理和执行、设置或读取连接属性等几大类别。

- PDO

```
__construct (string $dsn
             [, string $username
             [, string $password
             [, array $options]]])
```

这是 PDO 类的构造器，它将在你发出 new PDO()调用的时候执行。这个构造器将尝试连接一个数据库服务器。如果尝试成功，它将返回一个数据库句柄对象；如果它在执行时发生了一个错误，PHP 将抛出一个 PDOException 对象。

```
try
{
  $dbh = new PDO("mysql:host=localhost;dbname=sampdb", "sampadm", "secret");
}
catch (PDOException $e)
{
  die ($e->getMessage () . "\n");
}
```

把某给定数据库句柄设置为 NULL 将关闭连接：

```
$dbh = NULL;
```

$dsn 参数代表 DSN（data source name，数据源的名字）。DSN 可以有多种表示方式。

- 驱动器 DSN。以驱动程序名和一个冒号开头，后面是一些与该种数据库系统相关的可选参数。就 MySQL 而言，它要求 DSN 应该是下面这样：

  mysql:host=*host_name*;dbname=*db_name*

  host 参数用来指定服务器主机，dbname 参数用来指定默认数据库。默认的 host 值是 localhost。如果 dbname 参数被省略，其含义是暂不指定默认数据库。可能出现在这里的其他参数还有用来指定 TCP/IP 端口号的 port 参数和用来给出 Unix 套接字文件路径名的 unix_socket 参数。如果你已经给出了一个 unix_socket 参数，就用不着再给出 host 或 port 参数了。
- URI DSN。以 uri：开头，后面是一个 URI 标识符，沿该 URI 标识符给出的访问路径找到的文件里存放着一个驱动程序 DSN。这种 URI 既可以指向一个本地文件，也可以指向一个远程文件。下面是一个本地 URI 的例子：

  uri:file:///usr/local/lib/my-dsn-file
- 别名 DSN。这是对 php.ini 文件里 pdo.dsn.*XXX* 形式的配置参数的统称，这种配置参数里的“*XXX*”就是所谓的别名。比如说，别名 sampdb 对应着配置参数 pdo.dsn.sampdb，这

个参数在 php.ini 文件里的设置值应该是一个驱动程序 DSN。

- 可选的`$username`和`$password`参数用来给出一个 MySQL 账户的用户名和口令。
- 可选的`$options` 数组用来给出未在其他参数里设定的连接选项。下面列出的部分选项是 MySQL 驱动程序独有的，另外一些比较通用，其他数据库驱动程序应该也能支持。对于用来调控某种行为的开关型整数值选项，把它们设置为 1 或 0 将分别启用或禁用相应的行为。
- `PDO::ATTR_AUTOCOMMIT`（整数值；默认启用）。启用或禁用自动提交模式。
- `PDO::ATTR_PERSISTENT`（整数值；默认禁用）。允许或禁止使用永久性连接。
- `PDO::ATTR_TIMEOUT`（整数值；默认为 300）。对 MySQL 而言，这是以秒计算的连接倒计时等待时间。对其他数据库系统而言，这个属性可能另有含义。
- `PDO::MYSQL_ATTR_DIRECT_QUERY`、`PDO::ATTR_EMULATE_PREPARES`（整数值；默认启用）。允许或禁止使用直接语句。如果启用了这个属性，占位符的使用效果将是在客户端模拟出来的，然后将查询发送到服务器。
- `PDO::MYSQL_ATTR_INIT_COMMAND`（字符串值）。一条在与 MySQL 服务器建立连接后立即执行的语句，它也会在每次自动重建连接后立即执行。
- `PDO::MYSQL_ATTR_LOCAL_INFILE`（整数值；默认禁用）。启用或禁用 `LOAD DATA LOCAL` 语句。请注意，MySQL 服务器不一定支持 `LOCAL` 能力、PHP 安全模式也有可能正在生效。如果遇到这两种情况之一，即使你启用了这个属性也没有实际效果。
- `PDO::MYSQL_ATTR_MAX_BUFFER_SIZE`（整数值；默认为 1MB）。PDO 返回的数据列值的最大长度，以字节计算单位。超过这个长度的值将被截短。
- `PDO::MYSQL_ATTR_READ_DEFAULT_FILE`（字符串值）。强制 PDO 从本属性指定的选项文件而不是从默认的选项文件读取选项。
- `PDO::MYSQL_ATTR_READ_DEFAULT_GROUP`（字符串值）。从选项文件中由本属性指定的选项组读取选项。
- `PDO::MYSQL_ATTR_USE_BUFFERED_QUERY`（整数值；默认启用）。允许或禁止把查询结果集缓存在客户端。如果禁用了这个属性，每次只能从服务器取回一个数据行。

❑ `bool`

`beginTransaction` `(void)`

禁用自动提交模式并开始一个新事务。如果调用成功，这个方法将返回 `TRUE`；如果调用失败，则返回 `FALSE`。在结束事务的时候，调用 `commit()`方法将提交所有修改，调用 `rollback()` 方法将取消所有修改。

```
try
{
  $dbh->beginTransaction ();  # start transaction
  $dbh->exec ($stmt1);        # execute statements
  $dbh->exec ($stmt2);
  $dbh->commit ();            # commit if successful
}
catch (PDOException $e)
{
  # roll back if unsuccessful, but use empty
  # exception handler to catch rollback failure
  print ($e->getMessage () . "\n");
  try
  {
```

```
    $dbh->rollback ();
  }
  catch (PDOException $e) { }
}
```

- bool

  commit (void)

  提交本次事务并恢复自动提交模式。如果调用成功，这个方法将返回 TRUE；如果调用失败，则返回 FALSE。

  这个方法的使用示例见 beginTransaction()方法条目中的例子。

- string

  errorCode (void)

  为本数据库句柄上最近一次执行的操作返回一个由五个字符组成的 SQLSTATE 状态值字符串。如果返回值等于 PDO::ERR_NONE（"00000"），其含义是“没有错误”。

```
if (!($sth = $dbh->query ($stmt)))
{
  print ("The statement failed.\n");
  print ("errorCode: " . $dbh->errorCode () . "\n");
  print ("errorInfo: " . join (", ", $dbh->errorInfo ()) . "\n");
}
```

- array

  errorInfo (void)

  为本数据库句柄上最近一次执行的操作返回一个包含出错信息的数组。该数组有 3 个元素，它们的值分别是 SQLSTATE 状态值（与 errorCode()方法的返回值完全一样）、出错代码值和出错消息值。后两个值与当前使用的数据库驱动程序有关。就 MySQL 而言，出错代码是一个数值，出错信息是一个字符串。

如果最近一次执行的操作成功了，这个方法的返回值应该是一个只包含 SQLSTATE 值 PDO::ERR_NONE（"00000"）的单元素数组。

这个方法的使用示例见 errorCode()方法条目中的例子。

- int

  exec (string $statement)

  执行参数所给出的 SQL 语句并返回该 SQL 语句所影响的数据行的个数。如果发生错误，则返回 FALSE 或一个空字符串。

```
$count = $dbh->exec ("DELETE FROM member WHERE member_id = 149");
printf ("Number of rows deleted: %d\n", $count);
```

  exec()方法只适用于 INSERT 或 DELETE 等修改数据库内容的语句。对于那些会生成一个结果集的语句（如 SELECT 语句），应该调用 query()方法去执行它们。

- mixed

  getAttribute (int $attribute)

  返回指定的数据库句柄属性的值。如果调用不成功，则返回 FALSE。

  I.3.4 节列出了一些可以用这个方法去检索的属性。

```
printf ("Driver name: %s\n",
        $dbh->getAttribute (PDO::ATTR_DRIVER_NAME));
printf ("Server info: %s\n",
```

```
        $dbh->getAttribute (PDO::ATTR_SERVER_INFO));
printf ("Server version: %s\n",
        $dbh->getAttribute (PDO::ATTR_SERVER_VERSION));
```

- array
  getAvailableDrivers (void)

  返回一个数组，其内容是当前可用的所有 PDO 驱动程序的名字。

```
$drivers = $dbh->getAvailableDrivers ();
printf ("Number of drivers available: %d\n", count ($drivers));
print ("Driver names: " . join (" ", $drivers) . "\n");
```

  这个方法也可以作为一个静态方法在无需事先获得一个数据库句柄的情况下调用：

```
$drivers = PDO::getAvailableDrivers ();
```

- string
  lastInsertId ([string $name])

  返回最近一次生成的序列编号。这个方法的具体行为取决于当前使用的数据库驱动程序。就 MySQL 而言，它就是 C API 函数 mysql_insert_id()的返回值。有些驱动程序要求必须通过 $name 参数给出一个序列对象的名字（MySQL 不在此列）。

```
$dbh->exec ("INSERT INTO grade_event (date, category)
             VALUES('2008-11-01','T')");
printf ("New grade_event ID: %d\n", $dbh->lastInsertId ());
```

- PDOStatement
  prepare (string $statement
  　　[, array $options])

  对第一个参数所给出的 SQL 语句进行预处理并返回一个 PDOStatement 类型的语句句柄以用于该语句的后续操作。如果该语句预处理失败，则返回 FALSE。执行一条预处理语句的办法是用它的语句句柄来调用 execute()方法。

```
$sth = $dbh->prepare ("INSERT INTO absence (student_id, date)
                       VALUES (?, ?)");
$sth->execute (array (7, "2008-10-01"));
$sth->execute (array (18, "2008-10-03"));
```

  预处理语句允许包含占位符，占位符以位置格式或命名格式给出均可。必须在调用 execute()方法之前把真实的数据值绑定到占位符或是传递给 execute()方法。在 I.3.2 节，你可以在关于 bindParam()和 bindValue()方法的条目里找到更多的例子。

  如果给出了可选的$options 数组，prepare()方法将根据该数组里的“键字/键值”组合对新生成的语句句柄的属性进行设置。

- PDOStatement
  query (string $statement
  　　[, *fetch_mode_option*] ...)

  执行第一个参数所给出的 SQL 语句，并返回一个 PDOStatement 类型的语句句柄以用于访问结果集；如果发生错误，则返回 FALSE。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
while ($row = $sth->fetch ())
  printf ("%s %s\n", $row[1], $row[0]);
```

query()方法只适用于那些会生成一个结果集的语句（如 SELECT 语句）。对于 INSERT 或 DELETE 等修改数据库内容的语句，应该调用 exec()方法去执行它们。

在第一个参数后面给出的所有参数将被传递到 setFetchMode()方法，并由该方法对 query()方法所返回的语句句柄的取回模式进行设置。允许在这里使用的参数见关于 setFetchMode()方法的条目。取回模式还可以通过在 query()方法返回后直接调用 setFetchMode()方法或者通过把一个模式值传递给 fetch()方法的办法来设置。取回模式决定着 fetch()方法所返回的对象的类型。

你还可以在没有调用fetch()方法的前提下把query()方法所返回的PDOStatement对象当做一个遍历指针来使用：

```
foreach ($sth as $row)
  printf ("%s %s\n", $row[1], $row[0]);
```

- string

```
quote (string $str
       [, int $param_type])
```

对作为第一个参数传递进来的字符串里的所有特殊字符进行转义(按照当前驱动程序的要求)，并添加必要的引号后返回结果字符串。如果驱动程序不支持这个方法，则返回 FALSE。

```
$quoted_val1 = $dbh->quote (13);
$quoted_val2 = $dbh->quote ("it's a string");
```

可选的第二个参数用来表明第一个参数的数据类型，其默认值是 POD::PARAM_STR。参数类型值的详细清单见 I.3.4 节。

quote()方法不能正确处理 NULL 值，它将返回一个空字符串而不是一个括在引号中的单词“NULL”。如果你的数据值有可能是 NULL，建议你使用占位符并把数据值绑定到占位符。这样才有可能让 PDO 能够正确处理所有的特殊字符。

- bool

rollback (void)

回滚当前事务并恢复自动提交模式。如果调用成功，这个方法将返回 TRUE；如果调用失败，则返回 FALSE。

这个方法的使用示例见 beginTransaction()方法条目中的例子。

- bool

```
setAttribute (int $attr,
              mixed $value)
```

设置数据库句柄的属性。第一个参数给出某个属性的名字，第二个参数提供该属性的设置值。如果调用成功，这个方法将返回 TRUE；如果调用失败，则返回 FALSE。

I.3.4 节列出了一些可以通过 setAttribute()方法设置的属性。

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_WARNING);
$dbh->setAttribute (PDO::ATTR_AUTOCOMMIT, true);
```

I.3.2 PDOStatement对象方法

每个 PDOStatement 对象代表由数据库句柄方法 query()或 prepare()返回的一个语句句柄。语句句柄上方法的用途大致分为执行语句、访问语句元数据和结果集内容、把数据值绑定到预处理语句、把变量绑定到结果集。

- bool
```
bindColumn (mixed $column,
            mixed $var
            [, int $type
            [, int $len
            [, mixed $options]]])
```
把一个结果集数据列绑定到一个 PHP 变量，这将使得该变量在你以 PDO::FETCH_BOUND 取回模式取回一个数据行时被设置为该结果集数据列的值。如果绑定成功，这个方法将返回 TRUE；如果绑定失败，它将返回 FALSE。

$column 值是某个结果集数据列的编号（从 1 开始）或名字（区分字母大小写；以驱动程序所返回的名字为准），$var 值是将在每次数据行取回操作中与该结果集数据列的值相绑定的 PHP 变量。

可选的$type 参数用来表明数据列的数据类型，其默认值是 PDO::PARAM_STR。可以用在这里的参数类型值的清单见 I.3.4 节。

可选的$len 和$options 参数的用途和用法与 bindParam()方法里的同名参数一样。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
$sth->bindColumn ("last_name", $l_name); # specify column by name
$sth->bindColumn (2, $f_name);           # specify column by position
while ($sth->fetch (PDO::FETCH_BOUND))
  printf ("%s %s\n", $f_name, $l_name);
```

- bool
```
bindParam (mixed $column,
           mixed $var
           [, int $type
           [, int $len
           [, mixed $options]]])
```
把一个 PHP 变量绑定到预处理语句中的一个占位符。如果绑定成功，这个方法将返回 TRUE；如果绑定失败，它将返回 FALSE。此后，你只需在调用 execute()方法之前把一个数据值赋值给该变量就可以把该数据值绑定到那个占位符了。

$column 值是预处理语句字符串中的某个占位符的编号（从 1 开始）或名字（一个以冒号开头的名字），$var 值是将与该占位符相绑定的 PHP 变量。

$type 参数用来表明该占位符的数据类型，其默认值是 PDO::PARAM_STR。可以用在这里的参数类型值的清单见 I.3.4 节。如果某个占位符还与一个 INOUT 类型的存储过程参数相关联的话，还需要把该占位符的数据类型与 PDO 常数 PDO::INPUT_OUTPUT 进行 OR 运算(比如说，PDO::PARAM_STR | PDO::INPUT_OUTPUT)。

$len 参数用来设置该占位符的数据长度。如果某个占位符还与一个 OUT 类型的存储过程参数相关联，你就必须为它设置一个明确的数据长度。

$options 参数用来向驱动程序提供数据。

```
$sth = $dbh->prepare ("INSERT INTO absence (student_id, date)
                       VALUES (:id, :date)");
$sth->bindParam (":id", $student_id);
$sth->bindParam (":date", $date);
$student_id = 7;
$date = "2008-10-01";
$sth->execute ();
```

```
$student_id = 18;
$date = "2008-10-03";
$sth->execute ();
```

- bool

  bindValue (mixed $column,
  mixed $value
  [, int $type])

  把一个数据值绑定到预处理语句中的一个占位符。如果绑定成功，这个方法将返回 TRUE；如果绑定失败，它将返回 FALSE。该数据值将被用于下一个 execute()方法调用。

  $column 和$type 参数的用途和用法与 bindParam()方法里的同名参数一样。

```
$sth = $dbh->prepare ("INSERT INTO absence (student_id, date)
                       VALUES (?, ?)");
$sth->bindValue (1, 7);
$sth->bindValue (2, "2008-10-01");
$sth->execute ();
$sth->bindValue (1, 18);
$sth->bindValue (2, "2008-10-03");
$sth->execute ();
```

- bool

  closeCursor (void)

  释放与当前语句相关联的资源。如果调用成功，这个方法将返回 TRUE；如果调用失败，它将返回 FALSE。如果你打算再次执行一条语句，但当前与语句句柄相关联的整个结果集还没有取回，你就应该调用这个方法。(就 MySQL 而言，这个步骤并不是必需的，因为驱动程序能够在必要时获取尚未取回的结果集残留部分。其他驱动程序不一定具备这种能力。)

- int

  columnCount (void)

  返回刚才执行的那条语句所生成的结果集里的数据列的个数。如果那条语句没有生成结果集或是还没来得及执行，这个方法的返回值将是零。

```
$sth = $dbh->query ("SELECT * FROM president");
printf ("Number of columns in result set: %d\n", $sth->columnCount ());
```

- string

  errorCode (void)

  除作用目标是 PDOStatement 对象以外，这个方法在其他方面和 PDO 对象的 errorCode()方法完全一样。

```
if (!$sth->execute ())
{
  print ("Could not execute statement.\n");
  print ("errorCode: " . $sth->errorCode () . "\n");
  print ("errorInfo: " . join (", ", $sth->errorInfo ()) . "\n");
}
```

- array

  errorInfo (void)

  除作用目标是 PDOStatement 对象以外，这个方法在其他方面和 PDO 对象的 errorInfo()方法完全一样。

这个方法的使用示例见 errorCode()方法条目中的例子。

- bool

execute ([array $params])

执行一条预处理语句；如果执行成功，返回 TRUE；如果执行失败，则返回 FALSE。

如果该语句包含占位符，你应该在调用 execute()方法之前先把数据值绑定到占位符上，或者是把那些数据值直接传递给 execute()方法。这个方法的使用示例可以在关于 prepare()、bindParam()和 bindValue()方法的条目里找到。

- mixed

```
fetch ([int $fetch_mode
       [, int $cursor_orientation
       [, int $cursor_offset]]])
```

从结果集提取下一个数据行并返回；如果结果集里已经没有数据行可供提取，则返回 FALSE。如果给出了$fetch_mode 参数，这个方法将按该参数所指定的格式取回数据行；否则，数据行的格式将由语句句柄的取回模式决定。这个方法默认使用的取回模式是 PDO::FETCH_BOTH，但你可以调用 setFetchMode()方法或是在调用$dbh->query()方法获得语句句柄时向它传递一个取回模式值，由此来改变这个默认值。I.3.4 节列出了一些可供选用的取回模式值。

$cursor_orientation 和$cursor_offset 参数用来控制可卷动游标。这两个参数不适用于 MySQL，因为它根本不支持可卷动游标。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
while ($row = $sth->fetch ())
  printf ("%s %s\n", $row[1], $row[0]);
```

- array

```
fetchAll ([int $fetch_mode
          [, int $column_index = 0
          [, array $constructor_args]]])
```

把结果集里剩余的所有数据行返回为一个以这些数据行为元素的数组，为这些数据行确定取回模式的办法与 fetch()方法所使用的相同。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
$rows = $sth->fetchAll ();
foreach ($rows as $row)
  printf ("%s %s\n", $row[1], $row[0]);
```

如果取回模式是 PDO::FETCH_COLUMN，fetchAll()方法返回的数组将只包含由$column_index 参数指定的那一个结果集数据列的值。结果集数据列的编号从 0 开始。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
$first_names = $sth->fetchAll (PDO::FETCH_COLUMN, 1);
print (join (", ", $first_names) . "\n");
```

$constructor_args 参数用来给出一个定制的类构造器，有关细节见《PHP 使用手册》。

- string

fetchColumn ([int $column_index = 0])

从结果集中的下一个数据行提取一个数据列并返回之；如果结果集里已经没有数据行可供提取，则返回 FALSE。$column_index 参数用来表明应该提取哪一个数据列。结果集数据列的编号从 0 开始。如果需要从每个数据行取回多个数据列，就不要使用这个方法。

```
$sth = $dbh->query ("SELECT COUNT(*) FROM member");
```

```
printf ("Number of members: %d\n", $sth->fetchColumn (0));
```

- mixed
  fetchObject ([string $class_name
  [, array $constructor_args]])

  从结果集提取下一个数据行并把它返回为一个类实例。如果结果集里已经没有数据行可供提取或是发生了一个错误，则返回 FALSE。

  $class_name 参数用来给出结果对象的类名（如果没有给出这个参数，则以 stdClass 为默认类名）。$constructor_args 参数用来给出一个定制的类构造器，有关细节见《PHP 使用手册》。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
while ($row = $sth->fetchObject ())
  printf ("%s %s\n", $row->first_name, $row->last_name);
```

- mixed
  getAttribute (int $attr)

  返回当前语句句柄的给定属性的值；如果调用不成功，则返回 FALSE。MySQL 数据库系统的驱动程序不支持这个方法，因为不存在任何与 MySQL 有关的语句属性。

- mixed
  getColumnMeta (int $column_index)

  把一个给定结果集数据列的元数据返回为一个关联数组；如果该结果集数据列不存在，则返回 FALSE。$column_index 参数用来表明应该返回哪一个数据列的元数据；结果集数据列的编号从 0 开始。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
var_dump ($sth->getColumnMeta (0));
var_dump ($sth->getColumnMeta (1));
```

  这个方法会返回哪些信息取决于具体的驱动程序。在我编写本书时，MySQL 数据库系统的驱动程序所返回的数组里包含如表 I-1 所示的值。

表 I-1

名　字	值
native_type	数据列里的值的PHP数据类型
flags	描述数据列的各种属性的标志
table	包含着数据列的数据表（表达式的这一栏是空白）
name	数据列的名字
len	数据列的长度
precision	数据列的精度
pdo_type	数据列的类型（对应着一个PDO::PARAM_*XXX*形式的值）

- bool
  nextRowset (void)

  如果某个语句句柄有多个数据行集（rowset），用该句柄调用这个方法将前进到下一个数据行集。如果调用成功，返回 TURE；如果调用失败，则返回 FALSE。

  导致多个数据行集的原因是调用了一个会生成多个结果集的存储过程，或者执行了一个由多条

语句以分号相隔而构成的语句字符串。这类似于使用 C API 处理多个结果集的情况（请参阅 7.8 节）。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president LIMIT 5;
                     SELECT 1, 2, 3;
                     SHOW TABLES");
do
{
  $rowset = $sth->fetchAll (PDO::FETCH_NUM);
  if ($rowset)
  {
    $count = 0;
    foreach ($rowset as $row)
    {
      for ($i = 0; $i < sizeof ($row); $i++)
        print ($row[$i] . ($i < sizeof ($row) - 1 ? "," : "\n"));
      $count++;
    }
    printf ("Number of rows returned: %d\n\n", $count);
  }
} while ($sth->nextRowset ());
```

- int

  rowCount (void)

  返回刚才执行的那条语句所影响的数据行的个数。这个方法只适用于 INSERT 或 DELETE 等修改数据行的语句。如果想为 SELECT 等会生成一个结果集的语句统计受其影响的数据行个数，还是取回数据行以后再进行统计更稳妥——调用 rowCount()方法得到的计数值不能保证有意义。

- bool

  setAttribute (int $attr,
  　　　　　　　mixed $value)

  对当前语句句柄的给定属性进行设置。第一个参数是该属性的名字，第二个参数是该属性的值。如果调用成功，这个方法将返回 TRUE；如果调用不成功，则返回 FALSE。MySQL 数据库系统的驱动程序不支持这个方法，因为不存在任何与 MySQL 有关的语句属性。

- bool

  setFetchMode (int $fetch_mode
  　　　　　　　[, *fetch_mode_option*] ...)

  为当前语句设置数据行取回模式。如果设置成功，这个方法将返回 TRUE；如果设置不成功，则返回 FALSE。这个方法会影响 fetch()和 fetAll()等方法将如何返回数据行——如果你在调用它们的时候没有明确地给出任何取回模式参数。

```
$sth = $dbh->query ("SELECT last_name, first_name FROM president");
$sth->setFetchMode (PDO::FETCH_OBJ);
while ($row = $sth->fetch ())
  printf ("%s %s\n", $row->last_name, $row->first_name);
```

  I.3.4 节列出了一些可以用作$fetch_mode 参数值的取回模式。

  在$fetch_mode 参数的可取值当中，有几个还需要你向 setFetchMode()方法多传递几个参数。

  - setFetchMode (PDO::FETCH_COLUMN, int $column_index)

    含义：从结果集中的数据行提取一个数据列并返回。参见关于 fetchColumn()方法的条目。

■ setFetchMode (PDO::FETCH_CLASS, string $class_name, array $constructor_args)
含义：从结果集提取数据行并把它返回为一个类实例。参见关于 fetchObject()方法的条目。
■ setFetchMode (PDO::FETCH_INTO, object $object)
含义：把结果集里的数据行返回到一个已经存在的对象里，各结果集数据列将被映射到该对象的类属性上。

I.3.3 PDOException对象方法

在默认的情况下，PDO 只在 PDO 类的构造器（也就是说，当你调用 new PDO()连接一个数据库句柄的时候）和 PDO 类的其他方法在其返回值表明本次调用失败的时候才抛出一个异常。如果你在连接建立以后启用了 PDO 的异常处理机制，来自 PDO 扩展包的所有方法将都会在调用失败时抛出一个异常，而包含在 PDOException 对象里的出错信息就是这种异常的一种直观结果。

为了启用 PDO 的异常处理机制，你需要使用一个数据库句柄，如下所示：

```
$dbh->setAttribute (PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
```

PDO 支持 3 种出错模式值。

❑ PDO::ERROR_SILENT。除了设置出错信息，PDO 不做任何其他事情。这是默认的出错模式。
❑ PDO::ERROR_WARNING。类似于静默模式，但 PDO 在设置出错信息以外还会弹出一条警告消息。
❑ PDO::ERROR_EXCEPTION。PDO 在设置出错信息之后将抛出一个异常。

如果 PDO 异常处理机制已启用，我们在有错误发生时就可以通过调用“异常”对象的 getCode()和 getMessage()方法获得出错信息。

对异常的默认处理行为是中止出错脚本的运行。如果你想自己处理它们，就需要用到 try/catch 语法构造。你可以像下面这样在 catch 语句块里通过调用“异常”对象的方法获得出错信息：

```
try
{
  $sth = $dbh->query ("SELECT * FROM no_such_table");
}
catch (PDOException $e)
{
  print ("getCode value: " . $e->getCode() . "\n");
  print ("getMessage value: " . $e->getMessage() . "\n");
}
```

❑ integer
getCode (void)

返回一个由 5 个字符组成的 SQLSTATE 值，它包含出错代码。如果这个返回值等于 PDO::ERR_NONE（"00000"），其含义是“没有错误”。

❑ string
getMessage (void)

返回一个包含出错信息的字符串。

I.3.4 PDO常数

本节将介绍一些可以用在 PDO 方法(如数据库句柄对象上的 `getAttribute()`和 `setAttribute()` 方法）里的常数。这里只列出了那些比较具有代表性的常数值，完整的 PDO 常数清单可以在《PHP 使用手册》中的 PDO 章节里查到。

下面是通用数据库句柄属性。

- `PDO::ATTR_AUTOCOMMIT`
  当前的自动提交模式。
- `PDO::ATTR_DEFAULT_FETCH_MODE`
  数据行取回模式。(从 PHP 5.2.4 版开始才成为数据库句柄属性。)
- `PDO::ATTR_DRIVER_NAME`
  PDO 驱动程序的名字。
- `PDO::ATTR_ERRMODE`
  出错处理模式。关于这个属性的可取值的描述见 I.3.3 节。

下面是 MySQL 独有的数据库句柄属性。

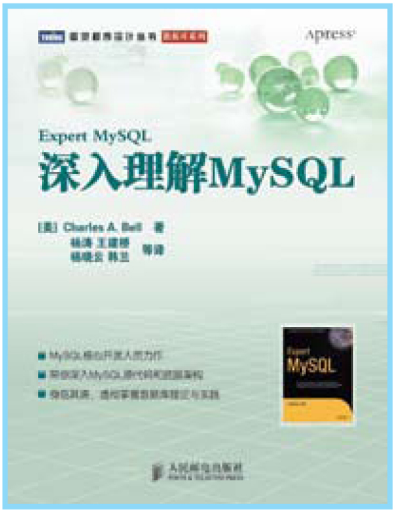
- `PDO::ATTR_CLIENT_VERSION`
  一个用来描述客户库版本的字符串。
- `PDO::ATTR_CONNECTION_STATUS`
  就 MySQL 而言，这个属性表明了连接是如何建立的。
- `PDO::ATTR_SERVER_INFO`
  一个用来提供一些服务器活动信息的字符串。
- `PDO::ATTR_SERVER_VERSION`
  一个用来描述服务器版本的字符串。

取回模式值，它们控制着结果集里的数据行将以何种形式被取回。

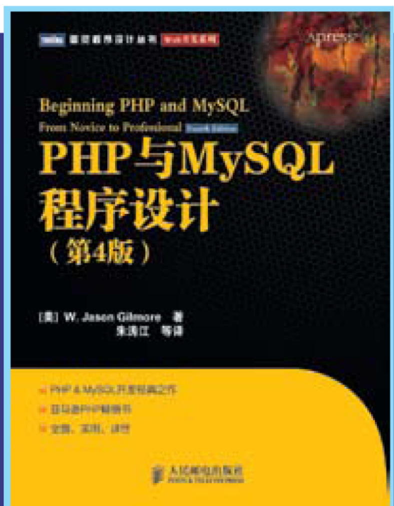
- `PDO::FETCH_ASSOC`
  返回一个数组，其元素需要使用关联下标来访问。
- `PDO::FETCH_BOTH`
  返回一个数组，其元素使用关联下标或数值下标都可以访问。
- `PDO::FETCH_BOUND`
  返回那些已提前通过调用 `bindColumn()`方法绑定到 PHP 变量的数据行元素。
- `PDO::FETCH_CLASS`
  把数据行元素返回到一个新建类实例的属性里。
- `PDO::FETCH_INTO`
  把数据行元素返回到一个现有类实例的属性里。
- `PDO::FETCH_NUM`
  返回一个数组，其元素需要使用数值下标来访问。
- `PDO::FETCH_OBJ`
  返回一个对象，其元素需要当做属性来访问。

下面是参数类型值。

- ❑ PDO::PARAM_BOOL
  一个布尔类型的参数。
- ❑ PDO::PARAM_INT
  一个整数类型的参数。
- ❑ PDO::PARAM_NULL
  代表 SQL 语言中的 NULL 值。
- ❑ PDO::PARAM_STR
  一个字符串类型的参数。


Apress
Expert MySQL
深入理解MySQL
[美] Charles A. Bell 著
人民邮电出版社


MySQL Crash Course
MySQL必知必会
[美] Ben Forta 著
Amazon全五星评价
人民邮电出版社


Apress
Beginning PHP and MySQL
From Novice to Professional
PHP与MySQL
程序设计
（第4版）
[美] W. Jason Gilmore 著
人民邮电出版社

“这是我读过的所有技术书中最好的一本，强烈推荐！”

——ACCU的*C Vu*杂志主编Gregory Haley

“本书是最权威的用户指南和参考手册，有了它，在MySQL数据库的日常操作和维护方面，你就会高枕无忧。”

——*Web Techniques*杂志主编Eugene Kim

MySQL Fourth Edition

MySQL技术内幕（第4版）

MySQL是一个开源关系数据库管理系统，现在越来越受到人们的青睐，应用范围也越来越广。MySQL的速度和易用性使其特别适用于Web站点或应用软件的数据库后端的开发工作。

本书是MySQL方面名副其实的圣经级著作，全面介绍了如何高效地使用和管理MySQL。书中主要介绍了如何将数据写入数据库，如何格式化查询，如何使用MySQL和PHP（或者Perl）生成动态网页，如何编写可以访问MySQL数据库的程序，如何管理MySQL服务器等。

Addison Wesley
www.informit.com

图灵网站：www.turingbook.com　热线：(010)51095186转604
反馈/投稿/推荐信箱：contact@turingbook.com
有奖勘误：debug@turingbook.com

分类建议　计算机/数据库/MySQL

人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-25595-2
定价：139.00元

欢迎加入

图灵社区

最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、审读、评选等多种活动，赢取积分和银子，积累个人声望。

ituring.com.cn